A Formal Approach to the Specification of Hierarchical Multi-Agent Systems

Proyecto Fin de Máster en Programación y Técnicas de Software
Master en Investigación en Informática
Curso 2008-2009

Autor: Carlos Molinero Brizuela
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Director: Manuel Núñez García

Autorizacion de difusion:

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "A Formal Approach to the Specification of a Hierarchical Multi-Agent System", realizado durante el curso académico 2008-2009 bajo la dirección de Manuel Núñez García en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen:

Este trabajo presenta una aproximación formal para crear especificaciones de sistemas de agentes jerarquizados. El funcionamiento permite una automatización de las tareas, a las que el usuario del sistema accederá a través de *peticiones*. De manera general, lo que se hace es definir los elementos básicos del sistema, los *agentes atómicos* que son los que están a cargo de realizar las transformaciones de los recursos, y el sistema recompondrá estos últimos para crear *agentes complejos*, que actúan como intermediarios entre los agentes atómicos y el usuario. Los agentes se localizan en una estructura de células, que permite la jerarquización. Las células conforman una estructura de árbol, y esta estructura conforma el sistema.

[agentes, sistemas multi-agente, jerarquización, automatización]

Summary:

This Master's Thesis provides a formal approach to create specifications of hierarchical systems of agents. The inner working of the system allows for an automatization of the tasks, that will be accessed by the user through the use of *petitions*. We will first define the main elements of the system, the *atomic agents* that are the ones in charge of executing the actual transformations of resources, and the system will recompose the latter to create *complex agents*, that will act as mediators in between the atomic agents and the user. The agents are localized over a structure of cells, that allows to create the hierarchical stratification. The cells conform a tree structure, and this structure conforms the system.

[agents, multi-agent systems, hierarchy, automatization]

Contents

List of Figures

Chapter 1

Introduction

The motivation of this work is to provide a flexible computational system capable of offering different solutions to new problems. Instead of pre-introducing thoroughly all the possible behaviors of a system, the system reorganizes its structure to be able to cope with new petitions coming from users. It bases itself in a series of behaviors (or tasks) that will be called *atomic* tasks which are the main bricks upon which to construct any new behavioral answer to the new needs that may appear.

The main skeleton of the system is constituted by this so-called *atomic* agents which are the ones that are in charge of executing the actual transformations that will occur in the system. Through the recombination of these *atomic* agents, new and more *complex* agents are created. The first advantage of this approach is that once the *atomic* agents are defined, little interaction from part of the user is needed. In addition, the lack of programming skills will also not be a disadvantage. All is needed is to specify the specific characteristics of the system, that is, what this system will be able to handle, by example procedures to call to the motors of a robot, and whatever future need of the user will be handled by the system itself.

Next we try to motivate how the recombination of our computational bricks will generate more complex structures. Let us imagine a robot with two main motors, one for its left wheel and another for its right wheel. Both of these motors are controlled by two atomic agents, one for each. In a first attempt to handle the system, we will ask the robot to modify its world by turning left. In this case the first atomic agent will act, creating no new real behavior, but if we ask the robot to move forward, the parallel acting of both atomic agents will be needed, and thus a new behavior is created through the acting of a complex agent.

The system is subdivided into a hierarchical structure in which agents are ordered con-

sidering its complexity (how many agents does it uses to perform a task) and also the *field of knowledge* to which that specific agent is inserted into. We assume here that if agents are located correctly into groups of related matters, meaning that if two agents perform similar tasks then they will be inserted together, when the pieces are recombined following the procedure that we explain later in this texts, they will be kept close in the hierarchical tree.

This hierarchical structure is implemented through a system of so called *cells* that conglomerate similar agents together. These cells are ordered in a tree structure, hanging all of them from the main cell. Every petition will be inserted into the main cell and will be carried on down through the tree until it has been processed. This also allows the possibility of facilitating a computational limit, meaning that the search could stop after it has descended a limit of levels down in the tree structure, if the system needs to take decisions in a certain amount of time that cannot be surpassed.

Currently there exists much discussion on what the term agent means in the Computer Science world. Since they are pieces of autonomous software, their definition can be ascribed to a great extent of objects. There are many different approaches defining agents, as well as different architectures being defined for the construction of multi-agent systems (MAS). Under this lines we present some of these definitions:

> *An agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments. These components are defined in a precise fashion, and stand in rough correspondence to their common sense counterparts. In this view, therefore, agenthood is in the mind of the programmer*, [Sho93].

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on ti, over time, in pursuit of its own agenda and so as to effect what it senses in the future*, [FG96].

> *An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives*, [Woo97].

According to [WJ95], next to the BDI (belief, desire, intention) mainstream architecture there exists yet another line of work in which properties like mobility, veracity, benevolence or rationality are considered. Further comments on this will be presented in the chapter of this Thesis devoted to related work.

In our work agents respond to petitions made by the user. These petitions modify the world since they create a new agent that will be able to perform the desired transformation of resources. They also start a communication protocol in between the agents, that will contain a broadcast message directed to every agent and replies, in the case that a specific agent can fulfill the petition contained in the broadcast message.

The system presented here, is composed of several agents that work together when needed to resolve a task. They communicate through a collection of resources. For example, this means that agentA asks agentB if it can incorporate 2 units of apples to the petition being handled; if so, agentB is adhered to the newly created complex agent.

Continuing with the review of how agents are perceived in Computer Science, agents are being advocated as a next generation model for engineering complex software systems (see for example [Jen00, WC01, BCP05] for an overview on agent-oriented software engineering). However, a number of questions about the nature and use of the agent-oriented approach are unanswered. Agents are flexible, they operate in cooperation, socialization, and their final outcome depends on this uncertain dialogue that may be established in the knowledge level of the agents (see [WJ95] for a review on the expected characteristics of agents). Still, methods to develop multi-agent systems in which we can assure reliability must be developed.

In order to enhance decision capabilities, it is very sensible to provide agents with appropriated and categorized information. In this line, analysis of various methods for knowledge based systems (see, for example, [SBD98]) led to the creation of a heuristic classification that was able to abstract a common problem solving behavior. Specific approaches of this heuristic classification are the *role limiting methods*, which are a shell to insert a specific implementation of a problem solving method, and *generic tasks*. The generic task method states that the structure and representation of domain knowledge is completely determined by its use (much as it happens in our approach, where concepts are only derived from the actual task that can be accomplished by atomic agents). The main problem underlying generic tasks is related to the predetermined problem solving strategy that they use. We are able to overcome this situation, since our methodology proposes a flexible problem solving strategy, that will vary depending on the kind of atomic agents included in the system.

If we try to incorporate the base of facts to a system, there will always be a lack of capacity to implement every possible structure of the agent, every different solution to the same problem, and every combination of small pieces that constitutes a complex problem. We believe it is easier and more feasible to incorporate bits of knowledge and, by having the system recomposing this information into complex tasks, solve a complex issue. This

approach simplifies at least two aspects. First, it helps to ensure the completion of the base of facts. Second, it allows to relocate the different agents, due to its modularity, so that they can be spread over a network to parallelize some of the tasks.

In this work, agents are treated conceptually as knowledge elements, agents are inserted as new abilities that the system can have access to, and then agents are created through the recombination of these abilities to generate complex tasks. When a petition is made to the system, it tries to accomplish it by reusing its bits of knowledge. In comparison with the human behavior, this is thought as a metaphor of the *mirror neurons* (see [RC04]), discovered by chance by the group formed by Giacomo Rizzolatti, Giuseppe Di Pellegrino, Luciano Fadiga, Leonardo Fogassi, and Vittorio Gallese at the University of Parma, Italy, while measuring the activities of neurons regarding the movement of a monkey. Unexpectedly some neurons fired not only when the monkey was moving the hand, but also when it looked how someone else was moving the hand. These led to a new theory of learning in which it is stated that they way humans and other mammals learn is through the neural mimic of activities seen in other individuals. Therefore, the how-to knowledge and the actual act is primarily fired by the same neurons.

In our approach all agents are located on a superstructure of *communication cells* created to record the hierarchy of the tasks/concepts and to be able to apply the framework on a distributed system. Each of this *cells* represents a macro-concept that all instances (*agents*) can be englobed into.

The system apprehends more complex concepts in the same way children do. First as a baby, one starts to involuntary move its muscles, noticing that that makes the arm move, with time one learns to control its arm, and the modifications that it performs in its surroundings. Afterwards, as more concepts and experiences (in our approach *petitions*) are incorporated, the baby no longer thinks about moving the arm but of reaching an object and grabbing it. That is, once we know how to perform an action the underlying mechanisms are automated; we no longer have to think about them.

In our last proposal, built upon previous work reported in [AMN08, AMN09], agents are presented as variant of Petri Nets, which are implicit inside each other (for a high level detail of Petri Nets see [Mur89]). There exist other approaches based on nested Petri Nets (see [Lom00, Lom04]). In those approaches Petri Nets are nested as the tokens of higher order Petri Nets. In our approach, agents have other agents' Petri Nets inside the connections from a place to a transition. Therefore, agents are launched as new threads when a place is left and in order to reach a transition the process of the nested agent must have been

completed.

The system may contain several ways to perform a certain petition, or even several agents that perform similar actions differing in the kind of resources that they consume in doing so. The problem of deciding how to act upon this situation is treated by using a threshold value for the *utility function*. The system measures the utility functions of all the agents, and using the maximal value, the others agents are compared to it using a threshold. If an agent falls below that threshold then it is discarded; in any other case, the system chooses to parallelize tasks.

Although much of the research based into the agent and multi-agent world is currently focused on ascribing human like ways of communication and representation of the world, and most of the related work presented on the next section of this master thesis will present those approaches, our approach focus on how the system behaves and to create a real implementation of it. We represent the world as a simple tuple that contains the resources available to the system. An upgrade of this work will surely have to focus on changing the representation of the world to a more realistic and complex *belief* system. Unfortunately the inner complexity of making this representational system behave as it does in theory, discourages anyone from actually making an implementation of it. In this respect, we will briefly comment on how examples of communications between agents described in research papers sometimes oversee the current possibilities of the AI world. We usually read in those papers expressions as <tell(agentA(open(door)))>. However, how does agentA know what a door actually is? Moreover what does it mean to open it? This is the reason why in this work communications and representations of the world are being simplified. Even so, the system implemented in this thesis could be the lower layer of a BDI MAS in which every agent is composed of a subset of agents with its inherent capabilities. Afterwards following the motivation, goals and believes of the agent, it self-creates petitions, thus forming new agents (agents of our system, that would be like a neuron of the agent of the higher BDI layer).

Chapter 2

Overview of some of the most relevant papers in the field

In this chapter we will go through some papers written in the topic of intelligent agents. They will vary from approaches based on BDI architectures to subsumption architectures. Some of them will fall into the how-to create the structures for the hierarchical system while some of them will assign roles to the agents, as well as to the multi-agents societies. We hope that this chapter allows to have an overview on the multi-agents research field.

2.1 George Kiss - Variable coupling of agents to their environment: Combining situated and symbolic automata

The paper [Kis91] addresses the problem that exist in multiple approaches of having to decide whether to express *generality* or *power* in the design of multi-agent systems. Generality is viewed as the ability of an agent to cope with changing and unexpected environmental conditions (flexibility), while power is thought of processing unit per time (effectiveness). There exist other requirements on agent designs, besides generality and power. Some of these requirements are *rationality*, *autonomy* and *reflexiveness*. Rationality means that agents actions are purposively appropriate, flexible and effective. In a BDI system this can be stated also as that an agent does not believe that something and its opposite are simultaneously true, and that the agent will act to satisfy its desires. Autonomy will be that an agent (thought as a process that runs continuously perceiving its environment) will react to changes in its environment in an autonomous fashion. Reflexiveness means that agents need to be aware of their capacities and their believes.

There is always a tradeoff in either giving a more flexible and general capacity to an agent or assigning it the biggest potential power of execution. Kiss thinks that layering is a valuable architectural solution to handle this tradeoff. He distinguishes three kind of layers:

- Fully deliberative actions.
  - Uses abstract and explicit world representations.
  - Each action is explicitly reasoned about.
  - Not automatic at all.
  - Most general.
- Complex skilled routines
  - Under strong stimulus control.
  - Some conscious decision making between alternatives.
  - Largely automatic.
  - Some narrow range flexibility.
- Simple reflex actions.
  - Direct stimulus, response links.
  - No conscious decision making at all.
  - Completely automatic.
  - Least general, completely situation specific.

The paper presents some mechanisms that may help enhance the power of a system, these mechanisms are:

- Specialization into modalities. For the sensors as inputs (depending on the kind of perception they produce) as well as for the outputs.
- Situatedness or coupling: Reactive mechanisms. Coupling means here the degree of constraint or controlling influence exerted by one process on another. Situatedness means that a system is coupled with its environment. Reactiveness is thought as a reflex-like procedure to outside stimulus.
- Non-symbolic, implicit representations.
- Trading space for time in computation.

- Parallelism.

In the same way the mechanisms for the enhancement of generality are:

- Decoupling. It achieves generality by providing a more constant internal situation.
- Dealing with the not-here and not-now. Meaning that we should create agents capable of dealing with future/past situations and possibilities.
- Symbolic, explicit representations.
- Non-modal representations.
- Use of small scale primitives.
- Seriality: Generality requires reusability, that is, advocate the use of sequential machines.

2.2 Anand S. Rao and Michael P. Georgeff - Modeling rational agents within a BDI architecture

The main contribution of this work [RG91] with respect to those of Bratman [Bra87] or Cohen and Levesque [CL90] is the inclusion of intentions as a main class element in the belief and desire architecture.

The authors use a possible world semantics. Modeling the world as a time tree. A particular point in that world is called a *situation*. Events transform one situation into another. There exists a distinction between primitive events (that transform the world into an adjacent point in the time tree) and non-primitive events, that can be viewed as planning and that transform the world to an non-adjacent point (being composed of several primitive events). Branches in the tree are choices.

The formalism distinguish between state formulas (evaluated at a time point in the tree) and path formulas (evaluated along a path). The operators used in the temporal logic used throughout the article are $\bigcirc$ next, $\diamondsuit$ eventually, $\Box$ always and $\cup$ until.

They use a notion of *strong realism* that means that for every belief-accessible world w, goals and intention accessible worlds are a subset of w at the specified time.

The authors use CTL which is a propositional branching time logic developed in [ES89]. The semantics is as follows:

Definition 2.1 An *interpretation* is a tuple $M =< W, E, T, \prec, U, \mathcal{B}, \mathcal{G}, \mathcal{I}, \Phi >$ where:

- W is a set of worlds.
- E is a set of primitive event types.
- T is a set of time points.
- $\prec$ is a binary relation on time points.
- U is the universe of discourse.
- Φ is a mapping of first order entities to elements in U for any given world and time point.
- $\mathcal{B}, \mathcal{G}, \mathcal{I}$ are the belief, goal and intention accessible worlds ($\mathcal{B} \subseteq W \times T \times W$).

□

Each *world* $w \in W$ called a *time tree* is a tuple $< T_w, \mathcal{A}_w, \mathcal{S}_w, \mathcal{F}_w >$ where:

- $T_w \subseteq T$ is a set of time points in w.
- $\mathcal{A}_w$ is the same as $\prec$ only restricted to T_w.
- $\mathcal{S}_w, \mathcal{F}_w : T_w \times T_w \mapsto E$ represent successful and failed events that occur between adjacent points.

In addition to this notations the paper also includes definitions for a sub-world (sub-tree of a world) and for a couple of extensions in the logic are given.

An agent has a belief ϕ (BEL(ϕ)) at time point t iff ϕ is true in all belief-accessible worlds of that agent at time t. A GOAL means that an agent has a desire to be in that situation. Formally GOAL(ϕ) holds iff ϕ holds in all goal-accessible worlds. Similarly, INTEND(ϕ) holds iff ϕ holds in all intention-accessible worlds.

Several axioms are also shown in the paper, Next, we present the ones that are more important:

- The *belief goal compatibility* states that if the agents adopt something as a goal, they believe that it is possible:

$$BEL(\phi) \subset GOAL(\phi)$$

- If something is an intention of an agent, then it is also a goal:

$$GOAL(\phi) \subset INTEND(\phi)$$

- Intention of action. If the agent believes to act upon a primitive event, then it does so:

$$INTEND(does(e)) \supset does(e)$$

- If an agent has an intention, then it believes that it has such an intention:

$$INTEND(\phi) \supset BEL(INTEND(\phi))$$

- If an agent has a goal, then it believes that it has such a goal:

$$GOAL(\phi) \supset BEL(GOAL(\phi))$$

- If an agent intends something, then it has a goal to intend it:

$$INTEND(\phi) \supset GOAL(INTEND(\phi))$$

- *Awareness of primitive events*:

$$done(e) \supset BEL(done(e))$$

- An agent will inevitably drop any intention:

$$INTEND(\phi) \supset inevitable\Diamond(\neg INTEND(\phi))$$

2.3 Franco Zambonelli, Nicholas R. Jennings and Michael Wooldridge - Organizational rules as an abstraction for the analysis and design of multi-agent systems

This paper [ZJW01] introduces three organizational concepts: *Organizational rules*, *organizational structures* and *organizational patterns*. The paper also introduces a formalism based on temporal logic, for specifying organizational rules. This approach uses previous approaches, such as Gaia [WJK00], to define multi-agent systems in terms of a role model. Agents are given a certain specific role that helps to define a structure.

The assigning of a role model, although useful for certain situations in which cooperate with each other, but to reach a higher level of generality the use of organizational structures proves useful. In words of the authors:

> *Organizational rules express general, global (supra-role) requirements for the proper instantiation and execution of a MAS. An organizational structure defines the specific class of organization and control regime to which the agents and roles have to conform in order for the whole MAS to work efficiently and according to its specified requirements. Organizational patterns express pre-defined and ubiquitous organizational structures that can be re-used from system to system (in a manner similar to the way catalogues of patterns are widely exploited in the design of object-oriented systems [GHJV93]).*

The authors continue by giving a definition of an autonomous agent:

> *Agents are software entities that exhibit autonomous and proactive goal-directed behavior -their activities are not subject to a global flow of control and they can take the initiative where appropriate and that are reactive to changes in the environment in which they are situated.*

Since agents usually work in a multi-agent system, with a number of agents working together or against each other to fulfill a goal, they act in a society of agents. Therefore, they exhibit a social behavior, interacting with one another. If we have an open system in which agents goals may enter in conflict, we need to define a social structure that allows the global desire goal to be fulfilled.

The use of an organizational metaphor can improve three aspects of the multi-agent system:

1. They help to characterize the role model for MAS.
2. They make the system less complex to manage and design.
3. When MAS are intended to support real world organizations, it reduces the conceptual gap between the software system and the organization they try to manage.

The use of *organizational structures* may serve as a way to group together agents that form a unified element, and reuse them in a more complex structure in which organizations and agents interact by exchanging knowledge or by coordinating their tasks with other agents.

Organizational rules help defining when a new agent should be accepted into the organization, and what role should it be assigned, which behaviors should be allowed inside the organization, and which should be prevented. *Organizational structures* work as a topology of the possible interaction patterns and the control regime of the organization's activities.

Organizational structures should be defined in the first place since they allow to define the role models. *Organizational patterns* would allow the reuse of several kinds of organizational structures. This will ease and speed-up the work of designers and developers. The idea is to create a catalog with the most useful and repeated structures.

Organizational rules specify relations and the possible interactions between different roles. The authors use a temporal logic to be able to define these rules, since rules are intrinsically temporal. The constructs of this logic include the following operators (besides all the normal logic operators):

- $\bigcirc\varphi$ means φ is true next.
- $\Diamond\varphi$ means φ is eventually true.
- $\Box\varphi$ means φ is always true.
- $\varphi\mathcal{U}\psi$ means φ is true UNTIL ψ is true.
- $\varphi\mathcal{W}\psi$ means φ is true UNLESS ψ is true.
- $\varphi\mathcal{B}\psi$ means φ is true BEFORE ψ is true.

Other elements added to the logic are:

- $plays(i, r)$ which means that agent i plays the role r.
- $card(r)$ is the cardinality of the agents that play role r.

Finally, the authors define the phases that would require the complete definition of a MAS following their organizational scheme, these phases are:

1. Definition of the organizational structure, by choosing the topology and the control regime. This involves considering the overall organizational efficiency, the corresponding real-world organization in which the MAS is situated, and the need to respect and enforce the organizational rules.
2. Completion of the preliminary role and interaction models, based upon the adopted organizational structure, and by keeping the organizational-independent aspects and the organizational-dependent ones as separate as possible.
3. Exploitation of well known organizational patterns in the design of the organizational structure and in the design of the final interactions model.

4. Definition of the agent model (as in Gaia). This identifies the agent types that will make up the system, and the agent instances that will be instantiated from these types. Here an agent type can best be thought of as a set of agent roles. There may in fact be a one-to-one correspondence between roles and agent types. However, this need not be the case. A designer can choose to package a number of closely related roles in the same agent type for the purposes of convenience.

5. Definition of the services model (as in Gaia). This identifies the main services that are required to realize the agent's role. A service is simply a single coherent block of activity in which an agent will engage. For each service that may be performed by an agent, it is necessary to document its properties. Specifically, we must identify its inputs, outputs, pre-conditions and post-conditions.

2.4 Pattie Maes - The dynamics of action selection

This paper [Mae89] addresses the problem of choosing an action in an autonomous multi-agent system. Actions are chosen following a rational goal oriented fashion. However, this approach can have conflicting goals, it should be adaptive to new situations, and there exists the possibility of a certain component failing, making it harder to reach the final goal.

This occurs in the situation of a mindless multi-agent system, such as those of Brooks' subsumption architectures. These systems, although desirable for properties such as modularity, distributed behavior, flexibility and robustness, lack a proper action selection procedure: Which agent should become active?. Moreover, what are the factors that determine a cooperation among certain agents?. The hypothesis assumed in this paper is that rational action of the global system can emerge and that there is not the need for "bureaucratic" agents (agents that decide which agent should become active).

There are several parameters that have to be tuned by the user, allowing to have different kinds of action selection procedures, such as more/less data oriented, goal oriented, deliberated, fast, etc.

Agents take part in a hierarchical system in the way that the activation of an agent is linked in a network of predecessor and successor links, which describe what agents should be activated before the current agent that is trying to perform an action. An agent is described by a tuple (l_p, l_a, l_d, a) where:

- l_p is a list of preconditions which have to be fulfilled before the agent can become active.

- l_a and l_d represent the post-conditions in terms of an add list and delete list scheme.
- a is the level of activation of the agent.

The links of the network are used to spread activation among agents belonging to it. When an agent's preconditions hold it spreads part of its activation level to its successors; otherwise, it augments the activation level of its predecessors. The algorithm that takes place at every time step is composed of the following steps:

1. The input from the state and goals to an agent is computed.
2. The spreading of activation of an agent is computed.
3. A locally computed *forgetting* factor ensures that the overall activation level remains constant.
4. The agents fulfilling the following 3 conditions become active: They have to be executable, their level of activation has to surpass a certain threshold and they must have a higher activation level than all other agents which fulfill the preconditions.

The parameters to be tuned in the system are:

- The threshold for becoming active.
- The percentage of their activation that is spread forward to other agents.
- The percentage of their activation that is spread backward to other agents.
- The relative amount of external input that comes from the goals as opposed to from the state of the environment.

2.5 Jose C. Brustoloni - Autonomous agents: Characterization and requirements

Brustoloni proposes in [Bru91] a classification of agents that is not based on the symbolic/reactive usual characterization but rather based on the amount of knowledge embedded in the system. This leads to a classification depending on the agents' ability to plan, adapt or reuse existing cases.

For Brustoloni, autonomous agents are systems capable of autonomous, purposeful action in the real world. They therefore must be reactive in some extent. This leads to the

problem of agents having to react in a timely manner (that is, that they should react fast enough to changes made in their environment). Thus putting too much stress in the system's computational capacities may lead to unusable agents.

In order to characterize the goal directed behavior expected from an autonomous agent, he proposes a system of drives, in a psychological sense, that each of the agents have. Attempts to fulfill these drives is what motivates agents to take one or another action. Actions and goals in his viewpoint must be *hierarchical*. Quoting from [Bru91]:

> *Actions and goals at one level exist only to accomplish goals at a higher level, and can generally be replaced by other actions and goals, which also would attain the higher level goal. At the bottom of this hierarchy are the primitive actions, the elementary actions directly supported by the architecture and from which more complex actions are composed*

In order to fulfill any goal, an agent must either have a pre-built knowledge of tasks and their results, or be able to find how to satisfy a drive by some sort of search directed behavior. Knowledge can be embedded structurally or symbolically. Structural knowledge allows for a faster respond to the environment, while symbolic knowledge is expected to have a more flexible behavior and higher possibility of achieving complex responses.

Next, Brustoloni makes the distinction between *regulation agents, planning agents and adaptive agents.*

- Regulation agents are those that do not do planning, they have all the knowledge required for their functioning inserted into their system. The main advantage of this kind of agents is their rapid response to the stimuli present in their environment. Obviously, the main drawback is that they are unable to show new behaviors emerged from a new situation in which they may be inserted.

- Planning agents are a higher kind of agents since they can also have knowledge of responses to the environment pre-built in their system. They count with tools to plan how to pursue new challenges. The main problem with these agents is that planning takes time, mainly when it is based on a pure search method to try to find the best possible plan. He distinguishes four different kinds planning agents. The first type will be a problem solving paradigm (uses search in the problem space to create a plan). The case-based paradigm (tries to find a similar case already handled or inserted into the system), faces the problem that yet to date there does not exist an efficient comparison algorithm (faster than pure search). Another kind of agent would be based

on various methods of operations research which do not handle well incomplete and noisy environment. Finally randomizing algorithms, which try to *naturally* find a suitable plan, without involving an extensive search, by using an appropriate heuristic. The main problem is that these agents do not always find the optimum solution.

- Adaptive agents are a special kind of agents that both learn about their primitive actions and the way to put them together in some sort of plan. They do not only face planning but also acquiring the knowledge required for their planning. However, this kind of agents seems to be more a possibility for future development than a reality in today's AI world. There exists a book in the subject [Dre93] that tries to apply Piaget's theory of how learning and general intelligence is achieved in neonates to the Computer Science world.

There is a hierarchy to relate all of these behaviors or types of agents. *Regulation agents* would account for instinct behaviors, a fast and reactive kind of behavior. *Case based agents* would be associated to habitual behaviors for which there already exist an outline of what to do. *Problem solving* agents would attain for a less frequent kind of necessity. This takes less time. Once a plan has been made, it would enter into the cases, making it unnecessary to develop again a plan, if the same situation would be repeated. *Randomizing algorithms* would be similar to playing, in which new ways of assembling primitives actions appear. Finally *adaptive agents* would be like theory making, very infrequent and requiring a long time to produce results.

2.6 Yoav Shoham - Agent-oriented programming

In his paper "Agent-oriented programming", Yoav Shoham [Sho93] proposes a new computational framework that promotes a societal view of computation, in which agents are combined to perform a certain computation. Agents are defined by their mental state, which is decomposed into beliefs, decisions, capabilities and obligations.

For Shoham an agent is:

> *An entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices and commitments. These components are defined in a precise fashion and stand in rough correspondence to their common sense counterparts.*

Clearly, it is a definition of the term agent biased towards its own work, since he is going to define precisely those elements as part of his agent programming language. Since the

possibility of ascribing beliefs, decisions, capabilities and obligations to any element, even the most simple of the systems, and therefore following its definition, considering that anything can be said to be an agent, he uses the words of John McCarthy [McC79], to explain when ascribing those mental states to a system is something useful:

> *To ascribe beliefs, free will, intentions, consciousness, abilities, or wants to a machine is legitimate when such an ascription expresses the same information about the machine that it expresses about a person. It is useful when the ascription helps us understand the structure of the machine, its past or future behavior, or how to repair or improve it. It is perhaps never logically required even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is most straightforward for machines of known structure such as thermostats and computer operating systems, but it is most useful when applied to entities whose structure is incompletely known.*

Shoham adopts the S5 modal logic (see [LL32]) which have properties that includes tautological closure, positive introspection and negative introspection. The semantics adopted are the possible world semantics.

He states that decisions are logically constrained, though not determined, by the agent's beliefs. These beliefs refer to the state of the world, to the mental state of other agents and to the capabilities of this and other agents. This perspective motivates the introduction of two mental categories: *Belief* and *decision* (or choice), and another not mental per se construct which is *capability*. In contrast, *decision* will be treated in terms of *obligation*, as an obligation to oneself.

The definitions of the mental categories are:

- Time: All operators are related to time.
- Belief: An agent a believes something on a certain time t: $B_a^t\varphi$
- Obligation: Agent a has an obligation to agent b on time t: $OBL_{a,b}^t\varphi$
- Decision: Agent a obliges itself: $DEC_a^t\varphi = OBL_{a,a}^t\varphi$
- Capability: Agent a is capable of doing something at time t: $CAN_a^t\varphi$

All of these constructs maintain a certain set of properties, like internal consistency of believes and obligations, good faith (agents only commit to what they believe themselves capable of), introspection (agents are aware of their obligations) and persistence of the mental state (agents have perfect memory of believes and obligations, and they only let go of a believe if they learn a contradictory fact).

Later on in the paper he discusses AGENT0, a language made to create agents and define its mental categories, and the message passing (communications) between agents. He also discusses the need for *agentification*, that is to create agent-like representation out of cameras or other devices so they can be used by agents.

2.7 Rodney A. Brooks - Elephants don't play chess

Rodney A. Brooks have written a series of articles talking about the subsumption architecture. One of them is "Elephants don't play chess" [Bro90] which is a summary of all the developments made by his team and a theoretical comparison with symbolic approaches.

He adheres himself to a current dogma in the AI world, called *situated activity*, based on the *physical grounding hypothesis*. This trend states that intelligence is a general property that can arise from the combination of a series of different reactive and *situated* (meaning that they only work located in a real world) agents that handle different parts of the overall robot's behavior.

His thesis against the symbolic approaches is that they are too field dependent, they are not capable of adapting to the noise existing in the real world, and that the sensory equipment is incapable of presenting with accurate symbolic descriptions of the objects that constitute the real world, rendering such approaches practically unusable. Also the number of calculations necessary to find solutions in the search spaces constitute another drawback of trying to use the symbolic position.

The physical grounding hypothesis works based on the assumption that every system needs to have its representations grounded in the physical world. Therefore, the connection of the system by sensors and actuators to the real world is the primary interest of these approaches. This kind of approach forces the construction of the system in a bottom up manner: Everything has to be concrete responses to the environment.

Another important point in the article is the explanation of the subsumption architecture. It is a way to program the robot based on incremental layers, each of them connecting perception to action, based on augmented finite state machines (AFSM). All of these layers are compiled to simulate parallelism. There are two subsumption languages, the old and the

new, used in different experimental robots. Next, we quote the general definition for the old subsumption language:

> *Each augmented finite state machine has a set of registers and a set of timers, or alarm clocks, connected to a conventional finite state machine which can control a combinational network fed by the registers. Registers can be written by attaching input wires to them, and sending messages from other machines. The messages get written into the registers by replacing any existing contents. The arrival of a message, or the expiration of a timer, can trigger a change of state in the interior finite state machine. Finite state machine states can either wait on some event, conditionally dispatch to one of two other states based on some combinational predicate on the registers, or compute a combinational function of the registers directing the result either back to one of the registers or to an output of the augmented finite state machine. Some AFSMs connect directly to robot hardware. Sensors deposit their calues in certain registers, and certain outputs direct commands to actuators.*
>
> *A series of layers of such machines can be augmented by adding new machines and connecting them into the existing network in a number of ways. (...)*

The new subsumption language uses behaviors, that are in fact AFSMs. The main tools to allow interactions between behaviors are message passing, suppression, and inhibition. Another difference with the old language is that behaviors can share registers, and that it provides a new more general timing mechanism than the original alarm clocks.

2.8 Stan Franklin and Art Graesser - Is it an agent or just a program?

In their paper [FG96], the authors try to address the problem of defining the meaning of the term *agent*, since it is applied to a great variety of themes and uses. They try to outline a possible taxonomy for autonomous agents.

The authors start with a brief summary of different cases in which the use of the word *agent* has been widely accepted by the scientific community. Among this cases we can find the MuBot agent, the AIMA agent, the MAes agent, KidSim agent, the Hayes-Roth agent, the IBM agent, the Wooldridge-Jennings agent, the SodaBot agent, the Forner agent, the Brustoloni agent and the FAQ agent.

The authors focus on defining an autonomous agent rather than giving a basic definition of agency and then building up to the concept of autonomy. Their definition is:

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, overtime, in pursuit of its own agenda and so as to effect what it senses in the future.*

They also present a discussion on how to describe an agent. Based on the definition of an autonomous agent, they present a series of elements to be enumerated when trying to describe an agent. In between those, they distinguish:

- Environment.
- Sensing capabilities.
- Actions.
- Drives (preferences).
- Action selection architecture.

In order to create a more useful classification of agents, they subdivide the possible agents affected by their definition into a series of subclasses depending on specific characteristics. These categories are:

- Reactive. Meaning that it responds in a timely fashion to changes in the environment.
- Autonomous. Exercises control over its own actions.
- Goal-oriented. They do not simply act in response to the environment.
- Temporally continuous. Continuously running processes.
- Communicative. They communicate with other agents.
- Learning. Changes in their behavior based on their previous experiences.
- Mobile. They are able to transport themselves from one machine to another.
- Flexible. Their actions are not scripted.
- Character. They present a believable personality and emotional state.

They also proposed a different taxonomy of agents, considering a biological classification. Following this line of thought autonomous agents are subdivided into *biological* agents, *robotic* agents and *computational* agents. The latter are subdivided into *artificial life* agents and *software* agents. Again this last category can be decomposed into *task-specific agents*, *entertainment* agents and *viruses*.

More information on these topics can be found in [WJ95] which is a well documented and extensive review on current agent theories, as well as in various AOSE reviews [Jen00, BCP05].

Chapter 3

A formal methodology to specify multi-agent systems

The proposed framework has been developed and presented in two papers. A preliminary version was accepted in the IEEE SITIS Conference [AMN08]. The version of our methodology presented in this chapter, a revision of the before mentioned paper, was presented in the ICCS Conference [AMN09]. This research will also be presented as a chapter in a book published by the organizers of the SITIS conference, containing revised and extended versions of selected papers, that will come out in the near future.

We introduce a novel methodology to formally specify complex multi-agent systems. Our approach allows us to redefine computational problems in terms of agents that perform certain tasks. In our view, a system is formed by the combination of atomic and complex agents. Atomic agents are in charge of executing atomic tasks while complex agents reunite and summarize the properties of their underlying atomic agents. Basically, our approach consists in specifying the smaller parts of the problem as atomic agents. Each atomic agent is in charge of executing a small transformation of resources. Afterwards, the system will recombine them to form complex agents that will embrace the knowledge of several atomic agents. All agents are located on a superstructure of communication cells created to record the hierarchy of the tasks and to be able to apply the framework on a distributed system. In order to provide a useful framework, we have developed a tool that fully implements all the stages of the methodology.

3.1 Introduction

Computational science embraces the concept of aiding the development of other studies in different fields through the use of new computational means. Therefore it has to create open systems that can be applied to a great extent of problems. In addition, it is relevant to take into account that the people to which it is focused are not, in general, computer scientists. Therefore, its easiness of use is a must. In this Master Thesis we report on a formalism that allows to solve complex problems through the use of agents. We propose a method to factorize the problem, being the first step to break down the problem into the smaller parts possible and assign an agent to each of those tasks. Then, the produced system allows to make petitions that will create other agents that, through recombination, are able to condense the information of several agents, so that they can solve a complex situation.

Even though there are general purpose formalisms to formally describe complex concurrent systems (such as process algebras and Petri Nets) they are not suitable to describe agents since these languages and notations do not provide specific operators to deal with the inherent characteristics of agents. However, there has been already several studies to formally describe the use of intelligent electronic agents that are nested into one another (see, for example, [Lom04, Lom08] for two approaches based on Petri Nets and automata, [NR01, NRR05a] for approaches based on process algebras, and [NRR05b, MNR07] for approaches based on finite state machines). Most of these approaches have been created in favor of comprehensibility. Therefore they facilitate to derive and apprehend new properties. However, due to its complexity, these formalisms are not supported by suitable user-friendly tools. Thus, the specification of a system is a task that cannot be carried out by somebody that is not a real specialist in formal methods.

Our approach is able to assimilate the systems that we are interested into a *common places* structure in which one is able to locate the rest of the structure from higher order points. If we use the subway lines as a metaphor, we only need to know the location of the different stations, but the exact location of that small fruit shop that we are trying to reach is bounded to the location of the closest metro station. Once we arrive to that particular metro station, we will check the neighborhood map so that we can find the shop; we do not need to know in advance all the local maps associated with all the stations of the network. This is how our systems will work: Once we have all the atomic agents, each time that a new complex agent, embracing the knowledge of several atomic agents, is created we will refer to this new agent when making subsequent calls to the system. In this line, we are able to *forget* how atomic actions are performed because we have a higher order element to which

we can call upon. In any case, even with a complex structure, atomic agents are still the ones that execute *real* tasks.

Using another metaphor we could say that our approach produces systems that are similar to economic structures in which there exist intermediate agents that give us the result of the transformation of resources as a final product. These agents, in a hidden way, contract the prime manufacturers that create these resource transformations. Another point in favor of our approach is that it allows us to have an unbounded growth (equivalently, subdivisions as small as needed) either by adding agents in between existing ones or by assigning new atomic agents to the system that we had before.

It is important to note that the way our systems are subdivided, in so called *communication cells*, facilitates their deployment in a distributed system in which one can obtain a perspective of variable magnitude of the global tasks. This holds as long as we keep the hierarchical structure of the ensemble.

The rest of the chapter is organized as follows. In Section 3.2 we introduce some auxiliary notation. Section 3.3 represents the bulk of the chapter. There we define the syntax of the proposed formalism, giving a running example of a system implemented with our tool. In Section 3.4 we briefly describe the technical details of the architecture of the tool developed to specify the systems.

3.2 Preliminaries

In this section we introduce some notation that will be used throughout the rest of the Thesis. First, since users have different preferences, in order to properly design agents the first step consists in expressing these preferences. In order to extract preferences from users several mechanisms have been presented in the literature (see [DJJT01, GHH01, HH03]). In this paper, preferences in a given moment will be given by a *utility function*. These functions associate a value (a utility measure) with each possible combination of resources a user could own. Alternatively, other mechanisms such as *preference relations* could be used (see e.g. [MWG95] for conditions to transform one of the possibilities into the other).

In order to manage resources we will denote them as elements of a vector $\bar{x}$. We consider a *special* resource to record the performance of the system. The time that it takes to complete the tasks of the system will also be considered as another resource. A vector of resources is a vector of real numbers in which each number denotes the total amount of a specific resource. Along this chapter we consider that n is the number of resources of the system.

Definition 3.1 Let $\bar{x} \in \mathbb{R}^n$ be a *vector*. We have that x_i represents the *i-th* component of $\bar{x}$. Let $\bar{x}, \bar{y} \in \mathbb{R}^n$ be two vectors. We write $\bar{x} + \bar{y}$ to denote the *addition* of $\bar{x}$ and $\bar{y}$. We say that $\bar{q}$ is the addition of $\bar{x}$ and $\bar{y}$ if for all $1 \leq i \leq n$ we have $q_i = x_i + y_i$.

We denote by $\bar{0} \in \mathbb{R}^n$ the vector having all the value components equal to zero. We write $\bar{x} \leq \bar{y}$ if for all $1 \leq i \leq n$ we have $x_i \leq y_i$.

A *utility function* is defined as any function $f^u : \mathbb{R}^n \to \mathbb{R}$. We denote the set of all utility functions by $\mathcal{F}$.

□

Intuitively, given a utility function f^u, we have that $f^u(\bar{x}) > f^u(\bar{y})$ means that $\bar{x}$ is preferred to $\bar{y}$. For instance, if we have $\bar{x} = (x_1, x_2)$ representing the first element of the resource vector the number of apples and the second element the number of oranges, $f_1^u(\bar{x}) = 3 \cdot x_1 + 2 \cdot x_2$, means that, for example, the agent is equally happy owning 6 apples or 9 oranges. Let us consider another agent whose utility function is $f_2^u(\bar{x}) = 1 \cdot x_1 + 2 \cdot x_2$. Then, both agents can make a deal if the first one gives 3 oranges in exchange of 4 apples: After the exchange both are happier. Alternatively, if x_2 represents the amount of money instead of oranges then the first agent would be a customer while the second one might be a vendor. Utility functions allow a great expressivity in preferences. For instance, $f^u(\bar{x}) = x_1 \cdot x_2$ denotes that variety is preferred. A usual assumption is that no resource is a *bad*, that is, if the amount of a resource is increased, so does the value returned by the utility function. Using a derivative expression, this property can be formally expressed as $\frac{\Delta f^u(x_1,\ldots,x_n)}{\Delta x_i} \geq 0$ for all $x_1, \ldots, x_n \in \mathbb{R}$ and all $1 \leq i \leq n$.

Next we introduce a collection of *identifiers* to be able to univocally identify cells, agents and paths in the system. In the next section, we will formally define these concepts.

Definition 3.2 Let w be a system (see Definition 3.8). The set of all possible systems is represented by $\mathcal{W}$. We denote by ID^{C} the set of cell identifiers that are assigned uniquely to each of the cells. The function **newIdCell** : $\mathcal{W} \to \mathrm{ID}^{\mathrm{C}}$ returns an unused identifier for the world w. We use a special identifier $\texttt{nill} \in \mathrm{ID}^{\mathrm{C}}$ to denote an empty cell. We denote by ID^{A} the set of agent identifiers that are assigned uniquely to each of the agents belonging to the system. The function **newIdAgent** : $\mathcal{W} \to \mathrm{ID}^{\mathrm{A}}$ returns an unused identifier for an agent. We denote by ID^{P} the set of path identifiers that are assigned uniquely to each of the paths. The function **newIdPath** : $\mathcal{W} \to \mathrm{ID}^{\mathrm{P}}$ returns a fresh identifier for a path.

□

3.3 Definition of the formalism

In this section we present our formal language to specify complete systems as well as all the agents taking part in them. The basic notion to define the behaviour of agents is a *transition*, that is, a transformation of resources carried out by a specific agent. *Atomic* and *complex* agents will both hold transitions as a way to accomplish tasks, but *only atomic agents* will actually perform the transformation of resources. A transformation of resources is represented by a tuple $\bar{z} \in \mathbb{R}^n$. Intuitively, a positive component of the tuple z_i denotes that the transformation produces z_i units of the *i-th* resource while a negative component z_j denotes that the transition consumes z_j units of the *j-th* resource.

Definition 3.3 A *transition* of the system is represented by the tuple $(\bar{z}, id_p)$ where $\bar{z} \in \mathbb{R}^n$ is the transformation of resources and $id_p \in \mathrm{ID}^{\mathrm{P}}$ identifies the path that is in charge of executing the transition. The set of all transitions is denoted by $\mathtt{TR}$. □

In the following definition we introduce the concept of *path*. A path is a sequence of transitions. They allow to specify the situation where a *complex* agent has to execute several consecutive tasks.

Definition 3.4 Let $tr_1, \ldots, tr_m \in \mathtt{TR}$ be transitions. Then, $p = < tr_1, \ldots, tr_m >$ represents the *path* conformed by them. We have that p_i denotes the *i-th* element of the path, that is, the transition tr_i. The set of all paths is denoted by $\mathcal{P}$. We denote the empty path by $<>$. □

Next we show how to represent *agents*. We can distinguish between *complex* and *atomic* agents. *Atomic* agents assume the responsibility of actually implementing tasks, while *complex* agents cluster and delegate in the ulterior ones to accomplish complex tasks and summarize the properties of the agents that are implicity inside of them.

Definition 3.5 An *agent* is a tuple $a = (id, ib, P)$ where $id \in \mathrm{ID}^{\mathrm{A}}$ is a unique identifier for this agent, $ib \subseteq \mathcal{M}$ is the input buffer where messages will be stored, and $P \subseteq \mathcal{P} \times ID^p$ is the set of annotated-paths defining the possible behaviours of this agent, being each path labeled with an identifier. Intuitively, the meaning of this set of annotated-paths is that this specific agent will achieve through any of this paths a similar global transformation of resources. In other words, every path takes him from the same initial state towards a similar final state, differing one from another in the kind of transformations that they perform.

We denote by $\mathcal{A}$ the set of all agents. We define the function $\texttt{VTr} : \mathrm{ID}^{\mathrm{P}} \rightarrow \mathcal{P}$ as follows. Let $P = \{(<tr_1^{\alpha}, \ldots, tr_m^{\alpha}>, \alpha), (<tr_1^{\beta}, \ldots, tr_{m'}^{\beta}>, \beta), \ldots\}$ be a set of annotated-paths. We define $\texttt{VTr}(\alpha) = <tr_1^{\alpha}, \ldots, tr_m^{\alpha}>$. We also define the function $\texttt{VA} : \mathrm{ID}^{\mathrm{P}} \rightarrow \mathrm{ID}^{\mathrm{A}}$ that returns the agent that performs this path.

Let $a = (id, ib, P)$ be an agent, and $P = \{(<tr_1^{\alpha}, \ldots, tr_m^{\alpha}>, \alpha), (<tr_1^{\beta}, \ldots, tr_{m'}^{\beta}>, \beta), \ldots\}$ be the set of annotated-paths of agent a. We define the function $\texttt{VP} : \mathrm{ID}^{\mathrm{P}} \rightarrow \mathbb{R}^n$ using the auxiliary function $\texttt{VPAux} : \mathcal{P} \times \mathrm{ID}^{\mathrm{A}} \rightarrow \mathbb{R}^n$ as $\texttt{VP}(\alpha) = \texttt{VPAux}(\texttt{VTr}(\alpha), \texttt{VA}(\alpha))$ being defined as:

$$\texttt{VPAux}(<>, id) = \bar{0}$$

$$\texttt{VPAux}(<tr_1, tr_2, \ldots, tr_n>, id) = \begin{cases} \bar{z} + \texttt{VPAux}(<tr_2, \ldots, tr_n>, id) & \text{if} \quad tr_1 = (\bar{z}, id_p) \wedge \\ & \qquad id = \texttt{VA}(id_p) \\ \texttt{VPAux}(\texttt{VTr}(id_p), \texttt{VA}(id_p)) + & \text{if} \quad tr_1 = (\bar{z}, id_p) \wedge \\ \texttt{VPAux}(<tr_2, \ldots, tr_n>, id) & \qquad id \neq \texttt{VA}(id_p) \end{cases}$$

An agent is *atomic* if it has only one path in its set of annotated-paths, that path is conformed only by a single transition, and itself is the agent in charge of executing the transition. Formally, $a = (id, ib, P)$ is an *atomic* agent if the following restrictions are fulfilled: $\mid P \mid = 1$, and there exists $p = (<tr_1>, id_a) \in P$ such that $tr_1 = (\bar{z}_i, id_p)$ we have $\texttt{VA}(id_p) = id_a$. □

During the rest of the paper we consider that agents use *messages* to communicate among them. The next definition introduces the different kinds of messages that can be sent.

Definition 3.6 A *message* is given by a tuple $(t, s, ob, \bar{r})$ such that $t \in \{\texttt{BROADCAST}, \texttt{REPLIES}, \texttt{START JOB}, \texttt{FINISHED JOB}\}$, denotes the nature of the message and $s \in \mathrm{ID}^{\mathrm{P}} \cup \{\textbf{null}\}$ is the path origin of the message. In some cases this path can have the value **null**. The next item, $ob \in \mathrm{ID}^{\mathrm{P}} \cup \{\star\}$ is the objective of the message: It can be a specific path of an agent, or a broadcast message. The last component, $\bar{r} \in \mathbb{R}^n$ represents a transformation of resources. We denote by $\mathcal{M}$ the set of all messages. □

Example 3.1 Let $id \in \mathrm{ID}^{\mathrm{A}}$ be an agent identifier, $p_1, p_2 \in \mathrm{ID}^{\mathrm{P}}$ be path identifiers, and $\bar{r}$ be a vector of resources. A message $m = (\texttt{BROADCAST}, \textbf{null}, \star, \bar{r})$ represents a broadcast message ($\star$) sent by a petition wanting to find an agent that accomplish the transformation induced by $\bar{r}$. If we have a message $m = (\texttt{REPLIES}, p_1, p_2, \bar{r})$; then m denotes the message from agent $\texttt{VA}(p_1)$ that offers the path p_1, that replies to agent $\texttt{VA}(p_2)$ to the petition of performing a certain task of the path p_2, and specifies the transformation of resources $\bar{r}$. If

we have a message $m = (\texttt{START JOB}, p_1, p_2, -)$, m now represents the message from agent $\texttt{VA}(p_1)$ which is performing the path p_1 for asking to start the job to the path p_2 of the agent $\texttt{VA}(p_2)$. Finally, if $m = (\texttt{FINISHED JOB}, p_1, p_2, -)$, then m is the message from agent $\texttt{VA}(p_1)$ to agent $\texttt{VA}(p_2)$ to indicate that the path p_1, which is a sub-path of p_2, has just finished. □

Cells are elements that serve as baskets of agents to reunite, organize, conglomerate and handle petitions as well as calls to the agents.

Definition 3.7 A *cell* is a tuple $(\mathcal{A}, id, \text{Sons}, \text{Father}, ib)$ where

- $\mathcal{A} \subseteq \text{ID}^{\text{A}}$ is the set of agents that belong to the cell.
- $id \in \text{ID}^{\text{C}}$ is a unique identifier for this cell.
- $\text{Sons} \subseteq \text{ID}^{\text{C}}$ is the set of identifiers of the sons of this cell. If $\text{Sons} = \varnothing$ then we are in a node cell.
- $\text{Father} \in \text{ID}^{\text{C}}$ is the identifier of the cell that is father of this cell. If Father=`nill` then we are in the initial cell, from which all other cells are defined.
- $ib \subseteq \mathcal{M}$ is the input buffer where messages will be stored.

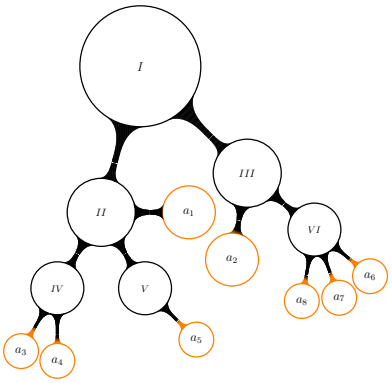
We denote by $\mathcal{C}$ the set of all cells. □

Next, we introduce the concept of *system* that contains in a tree like structure implicity defined by the father-son relationship, the cells that conform the whole system.

Definition 3.8 We say that a *system* (sometimes called *world*) is defined with a so called *origin cell* from where the tree of cells hang and by the vector of resources available in the system. Therefore, a *system* is a pair $w = (c, \bar{x})$ where $c \in \text{ID}^{\text{C}}$ is the origin cell, and $\bar{x}$ is the set of resources with which we deal in this world $\bar{x} \in \mathbb{R}^n$. We denote by $\mathcal{W}$ the set of all possible systems. □

We will use a simple running example to illustrate the previously introduced concepts.

Example 3.2 Let us consider that we have the world represented in Figure 3.1. As we observe in the figure, we have six cells, labeled from I to VI and eight agents distributed in them. For example, let us consider agent $a_3 = (id_3, ib_3, P_3)$. P_3 is the set of annotated-paths that this agent can perform, ib_3 represents the input buffer of this agent and id_3 is



$$
\begin{aligned}
a_1 &= (id_1, ib_1, P_1) \\
a_2 &= (id_2, ib_2, P_2) \\
a_3 &= (id_3, ib_3, P_3) \\
a_4 &= (id_4, ib_4, P_4) \\
a_5 &= (id_5, ib_5, P_5) \\
a_6 &= (id_6, ib_6, P_6) \\
a_7 &= (id_7, ib_7, P_7) \\
a_8 &= (id_8, ib_8, P_8) \\
W &= \{I, \bar{x}\}
\end{aligned}
\qquad
P_1 = \left\{ \begin{array}{lll} (<(\bar{z}_g, \kappa) & ,(\bar{z}_g, \mu)> & ,\alpha), \\ (<(\bar{z}_h, \kappa) & ,(\bar{0}, \nu)> & ,\beta) \end{array} \right\}
$$

$$
P_2 = \left\{ \begin{array}{lll} (<(\bar{z}_g, \varpi) & ,(\bar{0}, \varsigma)> & ,\gamma), \\ (<(\bar{0}, \varpi)) & ,(\bar{0}, \tau)> & ,\delta), \\ (<(\bar{0}, \upsilon) & ,(\bar{0}, \varsigma)> & ,\epsilon), \\ (<(\bar{z}_h, \upsilon) & ,(\bar{0}, \tau)> & ,\iota) \end{array} \right\}
$$

$$
\begin{aligned}
P_3 &= \{(<(\bar{z}_a, \kappa)>, \kappa)\} \\
P_4 &= \{(<(\bar{z}_b, \mu)>, \mu)\} \\
P_5 &= \{(<(\bar{z}_c, \nu)>, \nu)\} \\
P_6 &= \{(<(\bar{z}_d, \varpi)>, \varpi)\} \\
P_7 &= \{(<(\bar{z}_e, \varsigma)>, \varsigma)\} \\
P_8 &= \{(<(\bar{z}_f, (id_8, \upsilon))>, \upsilon)\}
\end{aligned}
$$

$$
\begin{array}{lllllllllll}
\bar{z}_a & = [& -50, & 0 & 0 & -40, & 0, & 0, & 0, & 1, & -20 \ ] \\
\bar{z}_b & = [& -100, & -300, & 0, & 0, & 1, & 0, & 0, & -1, & ,-30 \ ] \\
\bar{z}_c & = [& -80, & -450, & 0, & 0, & 1, & 0, & 0, & -1, & -40 \ ] \\
\bar{z}_d & = [& -200, & 0, & -300, & 0, & 1, & 0, & 0, & 0, & -20 \ ] \\
\bar{z}_e & = [& -50, & -10, & 0, & 0, & 0, & -20, & ,1, & 0, & -30 \ ] \\
\bar{z}_f & = [& -250, & 0, & -250, & 0, & 1, & 0, & 0, & 0, & -30, \ ] \\
\bar{z}_g & = [& -25, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & -20 \ ] \\
\bar{z}_h & = [& -30, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & -10 \ ]
\end{array}
$$

Position in the resource tuple	Meaning
1	**money**
2	**concrete**
3	**steel**
4	**wood**
5	**structure**
6	**bricks**
7	**wall**
8	**formwork**
9	**time**

Figure 3.1: Representation of a world.

the identifier of this agent. The set of annotated-paths P_3 contains a unique pair (pair, path identifier) $P_3 = \{(<(\bar{z}_a, \kappa)>, \kappa)\}$. The path identifier is κ the first element of the pair represents the chain of transitions that compose this path. In this case, the path is formed by a unique transition. This transition, $<(\bar{z}_a, \kappa)>$ represents that it is performed by the path κ of the agent $id_3 = \texttt{VA}(\kappa)$ and the exchange of resources after performing this transition is denoted by $\bar{z}$. This means that the resources of the world will change by applying $\bar{x} \leftarrow \bar{x} + \bar{z}_a$. In other words, it will generate a **formwork** unit, by using 50 units of **money**, 40 units of **wood**, and 20 **time** units.

For example, let us suppose that agent $a_1 = (id_1, ib_1, P_1)$ has two different annotated-paths. $(<(\bar{z}_g, \kappa), (\bar{z}_g, \mu)>, \alpha)$ and $(<(\bar{z}_h, \kappa), (\bar{0}, \nu)>, \beta)$. Next we explain one of these annotated-paths. The path identified by α, has two transitions in it. The first transition, denoted by the pair $(\bar{z}_g, \kappa)$, represents that this agent has to call to the annotated-path of agent $\texttt{VA}(\kappa)$ to perform it, and the transformation of resources by applying this transition is $\bar{x} \leftarrow \bar{x} + \bar{z}_g$. Then, after performing the complete α path the resources of the world would change to $\bar{x} \leftarrow \bar{x} + \bar{z}_a + \bar{z}_b$. Let us remember that the agent id_3 that is, the agent returned by $\texttt{VA}(\kappa)$ transformation function for the path κ is $\bar{z}_a$. □

All agents that are not *atomic* are *complex*. There are two ways to create agents. The first is to insert an atomic agent during the creation of the system. The other one is through *petitions* to the system, being the system in charge of recombining atomic and/or complex agents already embedded in the system to create a new complex agent.

Definition 3.9 A *petition* is a tuple $pet = (f^u, \bar{y}, \bar{o})$, where $f^u \in \mathcal{F}$ is a utility function, $\bar{y} \in \mathbb{R}^n$ is the vector of resources that is added to the resources already existing in the world, and $\bar{o} \in \mathbb{R}^n$ is the objective of the transitions, that is, the vector of resources that we expect to have after performing the petition. □

Intuitively, if we have a petition $pet = (f^u, \bar{y}, \bar{o})$, and $a = (id, ib, P)$ is the agent that has created the petition, if there exists $p \in \texttt{TR}$ such that there exists $(p, id_p) \in P$ then we have $\texttt{VP}(id_p) + \bar{x} + \bar{y} \geq \bar{o}$.

Example 3.3 We will explain the main types messages discussed before by applying a petition (a graphical representation is given in Figure 3.2). Let us consider a petition $pet = (f^u, \bar{y}, \bar{o})$. The tuple has three elements, the first one is a utility function (in this case $f^u = 10 \cdot x_1 + 5 \cdot x_9$), the second one is the set of resources added to the system, $\bar{y} = [0, 0, 0, 0, 0, 0, 0, 0, 1]$, and the third element of the tuple is the objective tuple of resources $\bar{o} = [0, 0, 0, 0, 1, 0, 1, 0, 0]$.

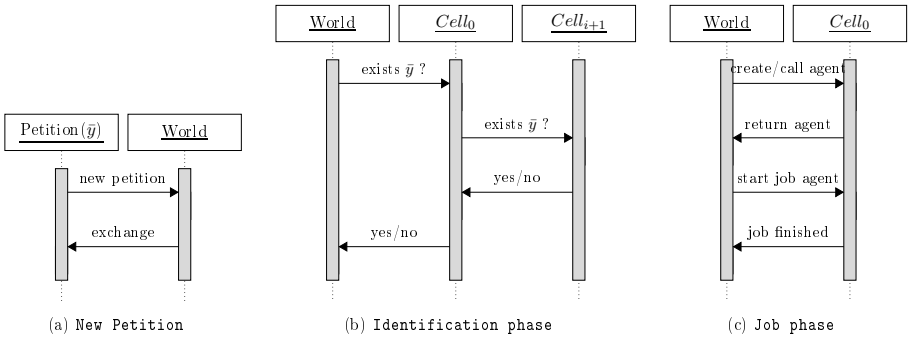


Figure 3.2: Schematic diagrams of world behaviour.

The first diagram of Figure 3.2 denotes that $pet = (f^u, \bar{y}, \bar{o})$ is inserted in the world $w = (I, \bar{x})$. When a new petition is inserted in the world, the resources of the petition are added to the existing vector of resources. After this initial stage, the world "asks" to its structure of cells if there are any agent(s) which can achieve the objective function $\bar{o}$.

□

3.4 Implementation

In this section we present our tool that facilitates the task of representing the different components of our framework. First, we are going to enumerate some of the technical requirements of the tool. Next, we will comment on some relevant parts of the implementation, and we will show how our simple running example can be represented.

The tool has been developed using J2EE Technology (Java, JDK 1.5, EJB) and the Netbeans software. It makes usage of the MVC architecture, to enable ease of maintenance. It also uses Java Swing components in order to develop Graphical User Interfaces(GUI).

The tool offers four different ways to create systems. The first one is by using an input XML-formatted file which contains all the description data of the system. The second way to input a system is by using the editor included in the GUI. The last way to create systems is by loading models saved previously. When a model is in the tool, it can be also saved by using a XML formatted file, in a database.

For the representation of the world, cells, and agents we have used *threads*. A java.lang. Thread object maintains the control for this activity. In fact, by representing each of the components by using threads we let the system represent a more realistic world. For example, agent a_1 can be waiting until agent a_3 has finished its task, while the world continues receiving petitions, and the cells continue forwarding messages between its agents.

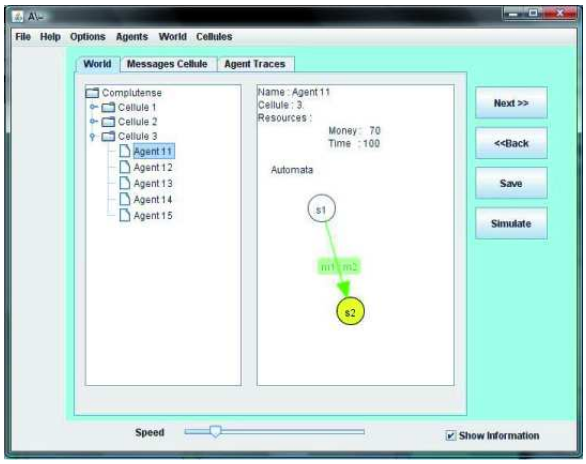


Figure 3.3: Phase 1 in the implemented tool

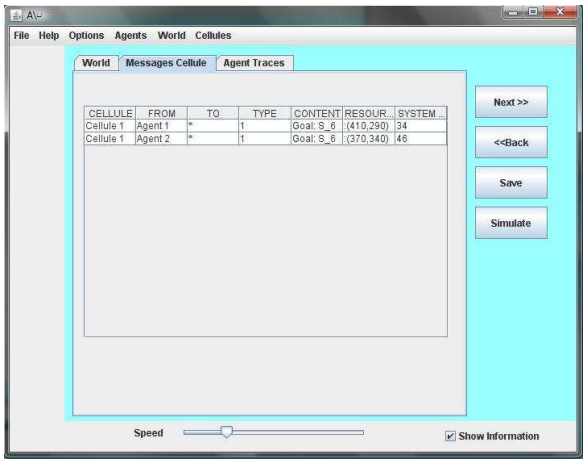


Figure 3.4: Phase 2 in the implemented tool

Another important task in a concurrency scheme is the management of shared memory, being the buffers implemented as circular buffers using a single, fixed size. Circular buffers are also used for data transfer between processes. The tool uses monitors in theses buffers to synchronize accessing threads. Conceptually, a monitor is a class whose data members are private and whose member functions are implicitly executed with mutual exclusion. In addition, monitors may define waiting conditions that can be used inside the monitor to synchronize the members functions. Figures 3.3, 3.4 and 3.5 show some screenshots from the tool.

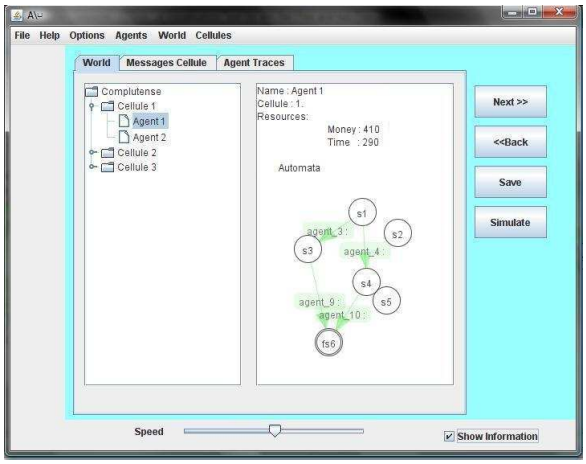


Figure 3.5: Phase 3 in the implemented tool

Chapter 4

The enhanced model

Since multiple modifications have been made to the original formalism, a chronological presentation of the formal model would make it hard to understand. Therefore, in this chapter we present the current framework without presenting the new advances separately. The main advantages with respect to the framework presented in the previous chapter are:

- Substitution of paths for Petri Nets as the way to model the behaviors of agents. This allows us to easily define the parallel execution of agents.
- Automation of the construction of the cell tree, through the computation of the least upper bound in terms of the cell tree.

In Section 4.1 we will present some introductory definitions. In Section 4.2 we will show the formal model. In Section 4.3 we will go thoroughly the steps to create a petition. In Section 4.4 we will show how an agent's Petri Net is executed. Finally, in Section 4.5 we will present a simple theoretical example that shows how the construction of the cell tree is performed.

4.1 Preliminaries

In this Section we briefly comment on the Petri Nets formalism and redefine the messages that we allow to be exchanged between agents.

Definition 4.1 The original definition of Petri Nets are tuples (P, T, W, M_0), where

- $P = \{p_1, p_2, \ldots, p_j\}$ is a set of places.
- $T = \{t_1, t_2, \ldots, t_i\}$ is a set of transitions.

- $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ assigns a weight to every connection between places and transitions and viceversa.
- $M_0 : P \to \mathbb{N}$ is the original marking, that is, the number of tokens assigned to each of the places in the beginning.

The execution of a Petri Net is defined as a modification on the marking. It is defined as follows:

- For all $t \in T$ we have

$$M \to_{G,t} M'$$

iff

$$\exists M'' : P \to \mathbb{N} : M = M'' + \sum_{p \in P} W(p,t) \ \wedge \ M' = M'' + \sum_{p \in P} W(t,p)$$

- $M \to_G M'$ iff there exists $t \in T$ such that $M \to_{G,t} M'$.

□

In this work, the function W is substituted by a simpler function F due to the fact that our connections can only have a weight of one (or zero if the connection does not exist). Another modification that apply to the F set is that the connection between a place and a transition includes a reference to another agent (using its identifier). In addition we have added a final places set, to decide whether the Petri Net has finished its execution. Finally, the way the Petri Net evolves is slightly modified:

Definition 4.2 A Petri Net in this work is a tuple $PN = (P, T, F, M_0, FP)$ where:

- $P = \{p_1, p_2, \ldots, p_j\}$ is a set of places.
- $T = \{t_1, t_2, \ldots, t_i\}$ is a set of transitions.
- $F : (P \times T \times ID^a) \cup (T \times P) \to \{0, 1\}$ assigns a weight of one (or zero if the connection does not exist) to every connection between places and transitions; and viceversa. It also adds an agent identifier to the connection between a place and a transition.
- $M_0 : P \to \mathbb{N}$ is the original marking, that is, the number of tokens assigned to each of the places in the beginning.
- $FP \subseteq P$ is the set of places that will be considered as final places, in order to know that the execution of the Petri Net is finished.

- Let $t \in T$ be a transition. We define the posset of t as $t\bullet = \{p_i \mid \exists f_i = (t, p_i) : F(f_i) = 1\}$. We define the preset of $t \in T$ as $\bullet t = \{p_i \mid \exists f_i = (p_i, t, -) : F(f_i) = 1\}$.

We define the evolution of the Petri Net as follows:

- For all $t \in T$ and agent identifier $id \in ID^a$

$$M \rightarrow_{G,t,id} M'$$

iff

$$\exists M'' : P \rightarrow \mathbb{N} : M = M'' + \sum_{p \in P} F(p, t, id) \ \wedge \ M' = M'' + \sum_{p \in P} F(t, p)$$

- $M \rightarrow_G M'$ iff there exists t and id such that $M \rightarrow_{G,t,id} M'$.

□

The following definition introduces the different kinds of messages that can be sent in our framework.

Definition 4.3 There exists two different kinds of messages, depending if they are used during the phase of the creation of the petition, or during the execution of the Petri Net.

- The first family corresponds to the BROADCAST and REPLIES identifiers.
- The second family corresponds to the STARTJOB and FINISHEDJOB identifiers.

We denote by ID^m the set of identifiers for messages.
A message from the first family is a tuple $m \in ID^m \times ID^t \times ID^a \times \mathbb{R}^n$ therefore if we have a message $m = (id_m, t_{act}, a_o, \bar{r})$ then:

- id_m denotes the nature of the message. We have $id_m \in \{BROADCAST, REPLIES\}$.
- $t_{act} \in ID^t$ is the identifier of the actual transition of the Petri Net that originated the BROADCAST MESSAGE, see Section 4.3 for an extended explanation (in this chapter ID^t will be used for identifiers of transitions).
- $a_o \in ID^a$ is the agent that originates the message.
- $\bar{r}$ is the tuple of resources that is needed in the BROADCAST message and the one that can supply the agent origin of the REPLIES message.

A message from the second family is a tuple $m \in ID^m \times ID^a \times ID^a$. Therefore, if we have a message $m = (id_m, a_0, a_f)$ then:

- id_m denotes the nature of the message. We have $id_m \in \{STARTJOB, FINISHEDJOB\}$.
- $a_0 \in ID^a$ is the agent origin of the message.
- $a_f \in ID^a$ is the agent objective of the message.

We denote by $\mathcal{M}$ the set of all messages.

□

4.2 Definition of the formalism

In this section we present our formal language to specify complete systems as well as all the agents taking part in them. *Atomic* and *complex* agents will both hold Petri Nets as a way to accomplish tasks, but only *atomic* agents will actually perform the transformation of resources. A transformation of resources is represented by a tuple $\bar{s} \in \mathbb{R}^n$. Intuitively, a positive component of the tuple denotes that the agent produces s_i unit of the *i-th* resource while a negative component denotes that the transition consumes s_j units of the *j-th* resource.

Next we show how to represent the *agents*. We can distinguish between *complex* and *atomic* agents. *Atomic* agents assume the responsibility of actually implementing tasks, and *complex* agents cluster and delegate in the ulterior ones to accomplish complex tasks. Agents have unique identifiers assigned. These identifiers can be seen as a word that denotes the concept that the agents represent.

Definition 4.4 An agent is a tuple $a = (id, ib, PN, \bar{s})$ where:

- $id \in ID^a$ is the agent identifier.
- $ib \subseteq \mathcal{M}$ is the input buffer.
- $PN = (P, T, F, M_0, FP)$ is a Petri Net.
- $\bar{s}$ is the overall transformation of resources that the agent accomplish. For an atomic agent this vector will be equal to the transformation induced by the connection from the first place to its transition and for a complex agent this vector will be the addition of the vectors of all the agents nested into itself.

Let us note that we do not store the order in which messages are received in the buffer. That is why we define a buffer as a set. We denote by $\mathcal{A}$ the set of all agents.

An agent is called atomic if it is in charge of executing a single task. Formally, we use a predicate $atomic : \mathcal{A} \mapsto \texttt{Bool}$, such that for all $a = (id_a, ib, PN, \bar{s}) \in \mathcal{A}$, if $PN =$

(P, T, F, M_0, FP), and $f_1 = (p_1, t_1, id_f)$, such that $F(f_1) = 1$, then a is an *atomic* agent if the following restrictions hold: $\mid P \mid = 2$, $\mid T \mid = 1$, and $id_f = id_a$

Let us remark that the notion of *atomic* agent means that the agent is itself in charge of executing the transformation of resources. □

Cells serve as baskets of agents to reunite, organize, conglomerate and handle petitions as well as calls to the agents. Abstractly, a cell is the macro-concept that holds the set of instances (agents) related in between them.

Definition 4.5 A *cell* is a tuple $(\mathcal{A}_{cell}, id, \text{Sons}, \text{Father}, ib)$ where

- $\mathcal{A}_{cell} \subseteq ID^a$ is the set of agents that belong to the cell.
- $id \in ID^c$ is a unique identifier for this cell. This can be seen as the concept that it represents.
- $\text{Sons} \subseteq ID^c$ is the set of identifiers of the sons of this cell. If $\text{Sons} = \varnothing$ then we are in a node cell.
- $\text{Father} \in ID^c$ is the identifier of the cell that is father of this cell. If Father=`nill` then we are in the initial cell, from which all other cells are defined.
- $ib \subseteq \mathcal{M}$ is the input buffer where messages will be stored.

We denote by $\mathcal{C}$ the set of all cells. □

Next, we define the whole system that contains in a tree like structure implicity defined by the father-son relationship, the cells that conform the whole system. This allows a hierarchical structuring of concepts.

Definition 4.6 A *system* is a tuple $w = (c_0, \bar{x}, \phi)$, where:

- c_0 the origin cell.
- $\bar{x} \in \mathbb{R}^n$ is the set of available resources in the system.
- ϕ is a threshold value that is used to discriminate between good and bad values of the utility functions.

□

All agents that are not *atomic* are *complex*. There are two ways to create agents. One is to insert an atomic agent during the creation of the system and the other is through *petitions* to the system, being the system in charge of recombining atomic and/or complex agents already embedded in the system to create a new complex agent.

Definition 4.7 Let $w = (c_0, \bar{x}, \phi)$ be a system, then a *petition* is a tuple $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$, where:

- $f^u \in \mathcal{F}$ is a utility function. of resources that this agent will create.
- $\bar{o} \in \mathbb{R}^n_-$ is the objective of the transitions, that is, the vector of resources that we expect to have after performing the petition.
- $\mathcal{A}_{pet} \subseteq ID^a$ is the set of agents capable of answering the petition. Initially this set is empty, and the petition fills it as it searches through the system.

We denote by $\mathcal{PET}$ the set of all petitions. We say that a petition $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ is fulfilled when:

$$\sum_{a_i \in A_{pet}} \bar{s_i^a} + \bar{o} + \bar{x} \geq \bar{0}$$

where for each agent in the set A_{pet} we have $a_i = (id, ib, PN, \bar{s_i^a})$. □

Next, we define a function GF that will be used to climb through the cell tree, until we reach a cell that has as father c_0.

Definition 4.8 Let $c = (\mathcal{A}_{cell}, id, \text{Sons}, \text{Father}, ib) \in \mathcal{C}$, and $w = (c_0, \bar{x}, \phi)$ be a system. We define $GF : \mathcal{C} \to \mathcal{C}$ as:

$$GF(c) = \begin{cases} c & \text{If Father} = c_0 \text{ or Father} = nill \\ GF(Father) & \text{Otherwise} \end{cases}$$ □

A petition deepens into the cell tree structure looking for a combination of agents capable of handling the needed transformation of resources. Subsequently, it creates an agent in the way described below.

Definition 4.9 The function $constrAg : \mathcal{PET} \to \mathcal{A}$, that creates an agent from a specific petition, is defined as follows. Let $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ be a petition. Then, $constrAg(pet) = (id, ib, PN, \bar{s})$, where id is a fresh agent identifier (created by the function $newIdAgent$) and ib is an empty new buffer. The resulting agent will have a connection between a place and a transition in PN for each agent in the set $\mathcal{A}_{pet}$ (for a detailed explanation on how to construct PN refer to 4.3). An invocation of this agent will generate a call to each of

those agents to execute their associated tasks. Finally the transformation of resources that this agent will accomplish is $\bar{s} = \sum_{a_i \in \mathcal{A}_{pet}} \bar{s_i^a}$, where for each agent in the set A_{pet} we have $a_i = (id_a, ib_a, PN_a, \bar{s_i^a})$. □

Before we insert the new agent into the tree structure, we must define how to compute the *least upper bound* of a set of agents.

Definition 4.10 The least upper bound (in short, lub) of a set of agents, given by a function $\bigsqcup : \wp(\mathcal{C}) \to \mathcal{C}$, is induced by the following order relation: $a \leq b$ iff there exist a descending path through the cell tree that goes from b to a. We define the lub as the lowest cell (in terms of the level in the tree) that remains a common path to reach all the cells in the set.

Insertion of an agent: Let $w = (c_0, \bar{x}, \phi)$ be a system and $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ be a petition, let $cells_{pet} = \{c \mid \exists c : c = (\mathcal{A}_{cell}, id, \text{Sons}, \text{Father}, ib) \in \mathcal{C} \ \wedge \ \exists a : a \in \mathcal{A}_{pet} \ \wedge \ a \in \mathcal{A}_{cell}\}$ be a set of cells, and $a_{new} = constrAg(pet)$ be the agent to be inserted. The a_{new} agent is inserted into a cell as follows:

- If there exists $\bigsqcup(cells_{pet})$ and $\bigsqcup(cells_{pet}) \neq c_0$, then insert a_{new} in $\bigsqcup(cells)$.
- Otherwise, let us consider the set $Fcells = \{GF(c) \mid c \in cells_{pet}\}$, we insert a_{new} into $c_{new} = (\{a_{new}\}, newIdCell(w), Fcells, c_0, ib)$ where ib is an empty buffer. In addition, for every element belonging to $Fcells$, change the father to be c_{new}.

□

4.3 Steps of a petition

Next we formally present how petitions are handled in our approach.

1. Let $w = (c_0, \bar{x}, \phi)$ be our system. Let $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ be our petition:
   - First we use a temporal vector of resources $\bar{z}$ that originally is assigned the value of $\bar{o}$, that is, $\bar{z} \leftarrow \bar{o}$.
   - The petition creates a temporal agent, through the use of $constrAg(pet)$, $a_{pet} = (id_{pet}, ib_{pet}, PN_{pet}, \bar{s}_{pet})$, where we have $PN_{pet} = (P, T, F, M_0, FP)$. The petition creates and inserts a place (p_0) in this newly created Petri Net, that is, $P = P \cup \{p_0\}$. This place will also be added to the final place set: $FP = FP \cup \{p_0\}$. It also creates and adds a transition t_0, $T = T \cup \{t_0\}$. Finally the petition creates a connection in between the transition and the place, that is, $F = F \cup \{((t_0, p_0), 1)\}$.

- We will also consider a special transition t_{act} initially assigned as t_0 that represents the actual transition to which the places in the next turn must be linked.

2. Next, the petition must discriminate depending on the number of negative resources in $\bar{z}$:

   - We consider a function $\varphi : \mathbb{R}^n \to \mathbb{N}$ that returns the number of negative resources.
   - If $\varphi(\bar{z}) = 1$, then the petition sends a message $m = (BROADCAST, t_{act}, \star, \bar{z})$ to the input buffer of every cell in the tree.
   - If $\varphi(\bar{z}) > 1$, then we would use another function $\chi : \mathbb{R}^n \to \wp(\mathbb{R}^n)$ that subdivides the resources from the petition in a collection of vectors of resources in which only one negative resource is allowed. This operation allows us to create parallel calling of the agents in the resulting Petri Net and sends the corresponding messages. As an example let us consider that $\varphi(\bar{z}) = 2$ and $\chi(\bar{z}) = \{\bar{z_1}, \bar{z_2}\}$. Then messages $m_1 = (BROADCAST, t_{act}, \star, \bar{z_1})$ and $m_2 = (BROADCAST, t_{act}, \star, \bar{z_2})$ will be sent simultaneously.

3. The cells retrieve the message from their input buffers (Choose(ib)) and retransmit the message in their input buffer to the agents that they withhold. So, for every agent in the set $\{a_i = (id_a, ib_a, PN, \bar{s}) \mid a_i \in \mathcal{A}_{cell}\}$ retransmit the message to its input buffer, that is, to ib_a.

4. The agents handle the message from the input buffer as follows: Let $a_i = (id_a, ib_a, PN, \bar{s})$ be the agent that is handling the message and let $m = (BROADCAST, t_{act}, \star, \bar{z})$ be the message being handled, if the agents internal transformation creates a resource that is negative in the resources of the petition, that is there exists i such that $s_i > 0, z_i < 0$ considering s_i as the i-*th* element of $\bar{s}$ and z_i as the i-*th* element of $\bar{z}$, then the agent sends back a message saying that he can fulfill the petition: $m = (REPLIES, t_{act}, a_i, \bar{s})$.

5. If the number of REPLIES messages with the same t_{act} is greater than one, then the petition uses the utility function to discriminate between the different possibilities. The way this is handled is by using a threshold value $0 < \phi < 1$, defined in the system $(w = (c_0, \bar{x}, \phi))$, we calculate the utility functions of all the agents involved and using the maximal value $max(f^u) = max(f^u(\bar{s_1^a}), \ldots, f^u(\bar{s_n^a}))$, where n is the number of agents that have answered with a REPLIES message: In the case where one agent's utility function is below the multiplication of this threshold by the maximal utility

function $f^u(\bar{s_i^a}) < \phi * max(f^u)$ with $0 < i \leq n$ then agent a_i is discarded from the set. All other agents are parallelized, as follows:

- We create a transition t_j.
- Add t_j to the transition set: $T = T \cup \{t_j\}$.
- Afterwards for every agent $a_i = (id_a, ib, PN, \bar{s})$, we do:
  (a) We update the set $A_{pet} = A_{pet} \cup \{a_i\}$.
  (b) We create a new place p_i with $0 < i \leq n$, where n is the number of agents
  (c) $P = P \cup \{p_i\}$.
  (d) $F = F \cup \{(p_i, t_{act}, id_a, 1), ((t_j, p_i), 1)\}$ where id_a the identifier of the agent.
  (e) We recalculate the resources needed by the petition, $\bar{z} \leftarrow \bar{z} + \bar{s}$.
  (f) We check if the petition is already fulfilled. Let our system be $w = (c_0, \bar{x}, \phi)$, then:
    – If $\bar{z} + \bar{x} \geq \bar{0}$, then we add a token in M_0 to the posset of t_j, that is, $M_0(t_j\bullet) = 1$. And stop adding agents. Delete t_i, that is, $T = T \backslash \{t_j\}$ and for every connection in F such that $f_i = (t_j, -)$ we set $F(f_i) = 0$. Afterwards we insert the agent in its cell, as explained under this lines.
    – Otherwise, we update $t_{act} \leftarrow t_j$. and re-send a message with the following information, $m = (BROADCAST, t_{act}, id, \bar{z})$.

6. Last, if the number of transitions that the Petri Net holds is one, that is, if $|\,T\,| = 1$ then we do not create nor insert an agent, we just call execution of the agent in charge of that transition. That is because, it means that there is already an agent capable of handling the petition in the system. In any other case, we calculate the lub of the cells that hold the agents that will be used by the agent create by the petition, that is, $\bigsqcup\{c \,|\, \exists c : c = (\mathcal{A}_{cell}, id, \text{Sons}, \text{Father}, ib) \in \mathcal{C} \ \wedge \ \exists a : a \in \mathcal{A}_{pet} \ \wedge \ a \in \mathcal{A}_{cell}\}$ as explained in its definition, and create a new cell if it does not exist. We insert the agent in the cell and execute it.

4.4 Execution of an agent

Let $a = (id, ib, PN, \bar{s})$ be the agent that we are executing, let $PN = (P, T, F, M_0, FP)$ be its Petri Net, we define the function $execute(id)$ as:

1. For all p such that $F(p, t, id_2) = 1 \ \wedge \ M_0(p) > 0$ perform the following steps:

(a) $\begin{cases} \bullet$If $id_2 = id$ then a is an *atomic* agent. Therefore, we transform the resources of the system: $\bar{x} \leftarrow \bar{x} + \bar{s}$. \\ \bullet Otherwise, we call upon $execute(id_2)$, by sending $m = (STARTJOB, id, id_2)$. \end{cases}$

(b) If either $id = id_2$ or there exists $m = (FINISHEDJOB, id_2, id) \in ib$, and there exists $t \in T$ such that $M_0 \rightarrow_{G,t,id_2} M_0'$, then we perform the transition so that $M_0 \rightarrow_G M_0'$.

2. Next, we check whether the execution of the Petri Net is finished.

   - If for all $p_i \in FP$, we have $M_0(p_i) = 1$, then we finish the execution and if there exists $m = (STARTJOB, id, id_2) \in ib$, then the system will send a message $m = (FINISHEDJOB, id_2, id)$.
   - Otherwise we return to step 1 and continue the execution.

4.5 Example of the construction of the cell tree

Next, we present a simple theoretic example to show the mechanism by which agents are inserted into the cell tree, and how these cells are created. Let us consider a robot with three motors. The first one allows the robot forward and backwards movements. The second motor allows the robot to control the wheel that provides the capability to turn. The third motor controls a robotic hand that opens and closes. We will insert these three atomic agents, that control each of the motors, so that the first two will be placed on a cell that represents the concept *moving* and the other one in another cell that represents *hand*. This is shown in Figure 4.1, up.

Next, we insert a petition that asks for a robot that moves to a certain place, picks up an object and then transports it elsewhere. Since no agent is capable of handling the petition, the system recombines them to be able to do it, creating a new agent. Because, in this case, the least upper bound is c_0, another cell has to be added and the relationships father/son have to be updated. This is shown in Figure 4.1, down. It is not relevant that *moving* and *hand* lose their father/son relationship with the main cell because, through recursion, they are still reachable as independent cells.

In order to make more clear the way agents are added, we will add another atomic agent that is in charge of turning on/off a torch that the robot may carry. The addition is done in a newly created cell. We can see it in Figure 4.2, up.

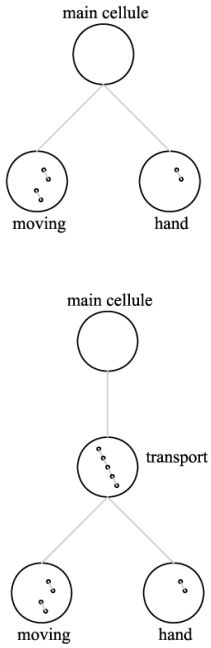


Figure 4.1: Construction of the cell tree. Phases 1 (up) and 2 (down).

Now, we ask the system to look for an agent that can turn on the torch and move to a certain point. The system recombines the existing agents and creates a new cell for the newly created agent, represented in Figure 4.2, down. Finally, we ask the system to turn on the torch and transport something. These updates are shown in Figure 4.3.

Let us remark that if another agent involving *moving* and using the *hand* is added, then it will be added in the *transport* cell since *transport* is equal to $\bigsqcup\{moving, hand\}$.

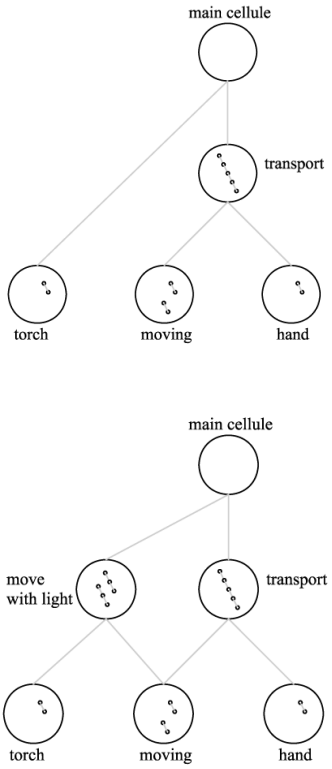


Figure 4.2: Construction of the cell tree. Phases 3 (up) and 4 (down).

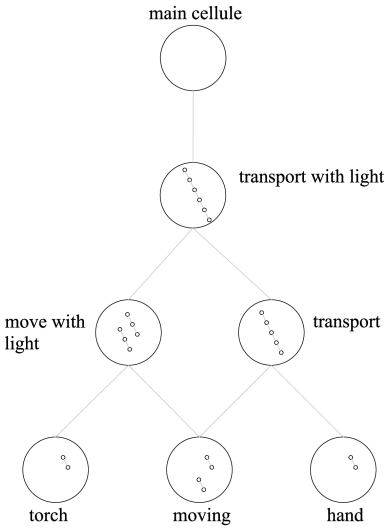


Figure 4.3: Construction of the cell tree. Phase 5

Chapter 5

Case study. Specification of a construction site.

This example will make use of all the capabilities of our framework. We will model a real world construction system This non-trivial system will allow us to understand better how our approach works to model complex systems.

5.1 Definition of the resources of the system

The system to complete the tasks needs to have a certain amount of resources, we will therefore begin by adding a collection of resources. In this case, they have been assimilated as raw materials. The resources that we are adding (and its places in the tuple of resources)are: time(1), money(2), concrete(3), metal(4), wood(5), and bricks(6).

The rest of the tuple of resources will be applied to the following concepts: terrain-movement(7), concrete-bed(8), site-ready(9), formwork(10), concrete-pouring(11), structure-finished(12), masonry-facade(13), carpentry-windows(14), facade-finished(15), interiors(16), finishes(17), and built-house(18).

Time is the only special resource, in the sense that the amount that we add at the start is the amount of time in which we want the petition to be fulfilled.

5.2 Definition of the *atomic* agents

The first step will be to create the tree that contains the cells that will locate the agents. The system has already created the cell c_0 and we will add 7 sons to it, to be able to classify

the different types of atomic agents correctly.

In cell c_1, that will be used to confine agents related to *terrain movement* we add agent $a_1 = (1, ib_1, PN_1, \bar{s_1})$, where $PN_1 = (P_1, T_1, F_1, M_1, FP_1)$, where $P_1 = \{p_0, p_1\}$, $T_1 = \{t_0\}$, and there exists $f_1 = (p_0, t_0, 1)$ and $f_2 = (t_0, p_1)$, such that $F(f_1) = 1$, $F(f_2) = 1$ and the value of F for any other connection is 0, $M_1 = \{(p_0, 1), (p_1, 0)\}$, $FP_1 = \{p_1\}$ and $\bar{s_1}$ is shown on the table below. This tuple means that the identifier of the agent is 1, which is the same of the identifier in charge of the transition (therefore, it is an atomic agent), when executed will consume 10 units of time and 100 units of money and create a unit of terrain-movement. The rest of the agents definition are similar and will be shown in the following table (all of the agents are atomic), including information about the cell where they are inserted and what resources will they consume and produce.

$CELL$																				
c_1	$\bar{s}_1$ = [	−10,	−100,	0,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0	]
c_2	$\bar{s}_2$ = [	−8,	−50,	−100,	0,	0,	0,	−1,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0	]
c_3	$\bar{s}_3$ = [	−2,	−10,	0,	0,	−100,	0,	0,	0,	−0.5,	1,	0,	0,	0,	0,	0,	0,	0,	0	]
c_3	$\bar{s}_4$ = [	−2,	−10,	0,	0,	−100,	0,	0,	0,	−0.5,	1,	0,	0,	0,	0,	0,	0,	0,	0	]
c_4	$\bar{s}_5$ = [	−30,	−120,	−100,	0,	0,	0,	0,	0,	0,	−2,	1,	1,	0,	0,	0,	0,	0,	0	]
c_5	$\bar{s}_6$ = [	−10,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0,	0,	0,	0	]
c_5	$\bar{s}_7$ = [	−10,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0,	0,	0,	0	]
c_5	$\bar{s}_8$ = [	−10,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0,	0,	0,	0	]
c_5	$\bar{s}_9$ = [	−7,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0	]
c_5	$\bar{s}_{10}$ = [	−7,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0	]
c_5	$\bar{s}_{11}$ = [	−7,	−60,	0,	0,	0,	−100,	0,	0,	0,	0,	0,	0,	0,	0,	−0.3,	1,	0,	0	]
c_6	$\bar{s}_{12}$ = [	−4,	−70,	0,	−100,	0,	0,	0,	0,	0,	0,	0,	0,	−3,	0.5,	0.5,	0,	0,	0	]
c_6	$\bar{s}_{13}$ = [	−4,	−70,	0,	−100,	0,	0,	0,	0,	0,	0,	0,	0,	−3,	0.5,	0.5,	0,	0,	0	]
c_7	$\bar{s}_{14}$ = [	−2,	−30,	0,	0,	−20,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	−2,	0.3,	0.3	]
c_7	$\bar{s}_{15}$ = [	−2,	−30,	0,	0,	−20,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	−2,	0.3,	0.3	]
c_7	$\bar{s}_{15}$ = [	−2,	−30,	0,	0,	−20,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	−2,	0.3,	0.3	]

Let us note that there are several columns that do not have a positive value. These resources, like structure-finished, are simply changes of connotation in the meaning of the transformations that will be carried out by complex agents. The insertion of these agents is represented in Figure 5.1.

5.3 First petition: Prepare the site to be built

For this first step we will introduce a petition in the system that prepares the site to be built. We consider a petition $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ where:

- $f^u = 2 * x_1 + 1 * x_2 + .3 * x_3 + 1 * x_7 + 1 * x_8$ is the utility function (all values $0 * x_n$ are not represented).

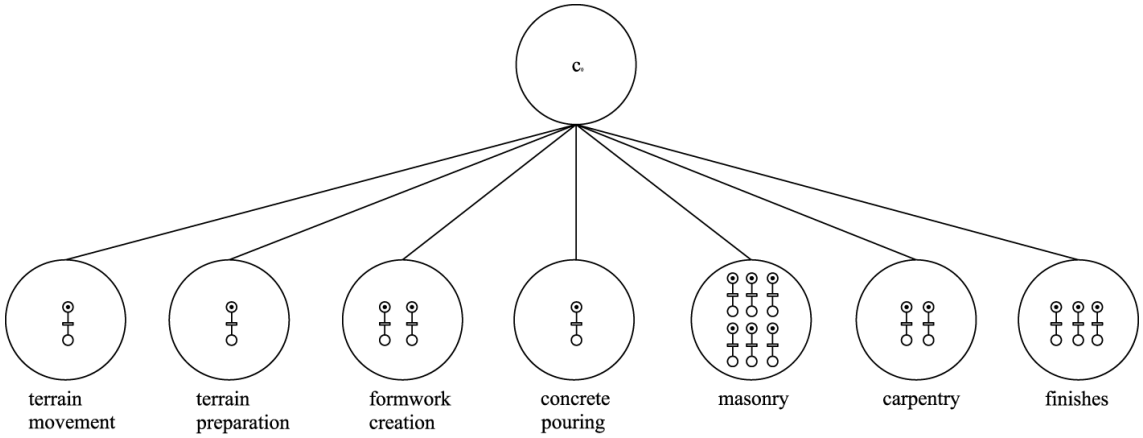


Figure 5.1: Insertion of the agents in the cell tree.

- $\bar{o} = (0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$.
- $\mathcal{A}pet = \{\}$ is the set of agents that are used by the petition (empty in the beginning).

The system sends the message with the petition and finds a_2 that supplies one resource where the negative value of the petition is. Thus adds agent a_2 to the petition so that $\mathcal{A}_{pet} = \{a_2\}$. Next, it creates agent a_{17}, adds a place, a transition connected to that place, and another place that connects to the transition in which agent a_2 is referenced to be executed in its turn.

Now, agent a_2 has added a new negative value in another position of the resources tuple. The petition sends a new message and agent a_1 replies. The petition adds agent a_1 so that $\mathcal{A}_{pet} = \{a_2, a_1\}$, adds another transition and another place with a connection that references a_1. Since now $resources \geq \bar{0}$, the petition finishes and adds a token to the last place created. We compute the $\bigsqcup\{c_i \mid c_i \in \mathcal{C} \ \wedge \ a \in \mathcal{A}_{pet} \ \wedge \ a \in \mathcal{A}_{c_i}\}$, since it does not exists one, it creates a new cell, and inserts agent a_{17} in it. The transformation of resources from agent a_{17} is $\bar{s_{17}} = \sum_{a \in \mathcal{A}_{pet}} \bar{s_a}$, such that $a = (id, ib, PN, s_a)$.

The result can be observed in Figure 5.2

5.4 Second petition: Create structure

Now we insert a new petition into the system, that constructs the structure of the building. We consider a petition $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ where:

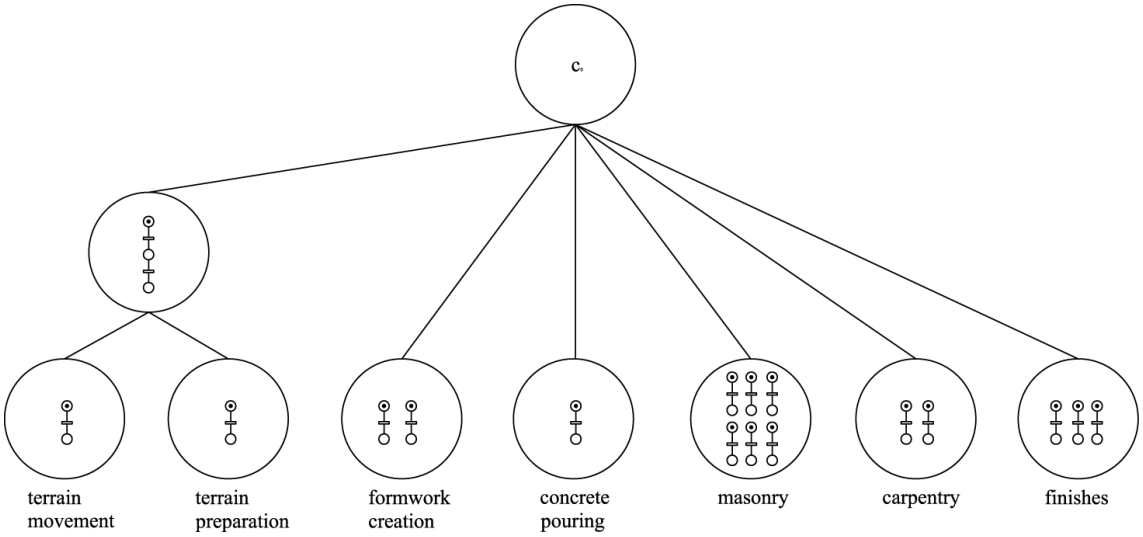


Figure 5.2: Creation of the first complex agent: Site is ready to build.

- $f^u = 2 * x_1 + 1 * x_2 + .3 * x_3 + 1 * x_{10} + 1 * x_{11}$ is the utility function (all values $0 * x_n$ are not represented).
- $\bar{o} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0)$.
- $\mathcal{A}pet = \{\}$ is the set of agents that are used by the petition (empty in the beginning).

The system behaves in a similar way to the preceding petition, so we will omit some details in the explanation of this petition, taking more time in the points that are different. First, we find agent a_5, and re-send the petition after adding its resources. Next, we find two similar agents that are able to respond to the new resource needed. Since both agents have identical utility function, it does not choose between them, and uses both of them in parallel to handle the petition quicker. If one of the agents would have had a *bad* utility function, that is, if $f^u_{a_i} < \phi * max(f^u)_{a_3,a_4,a_5}$, then the system would have had to decide if it was under a certain threshold, and that it was better not to use it. Since in this case all utility functions fulfill $f^u_{a_i} \geq \phi * max(f^u)_{a_3,a_4,a_5}$, they are parallelized. Next, we create two places with two connections that start in parallel and hold agents a_4 and a_3. Then, we re-send the petition and use the just created *complex* agent a_{17} as a starting point. The transition that holds agent a_{17} in reality will start the Petri Net from it, that will then call agents a_2 and a_1 although it appears as just one connection between a place and a transition in the Petri Net from the newly created agent a_{18}. The petition now looks for the lub of the cells. Since it does not find it, it creates a new cell and inserts agent a_{18}. Again the transformation of

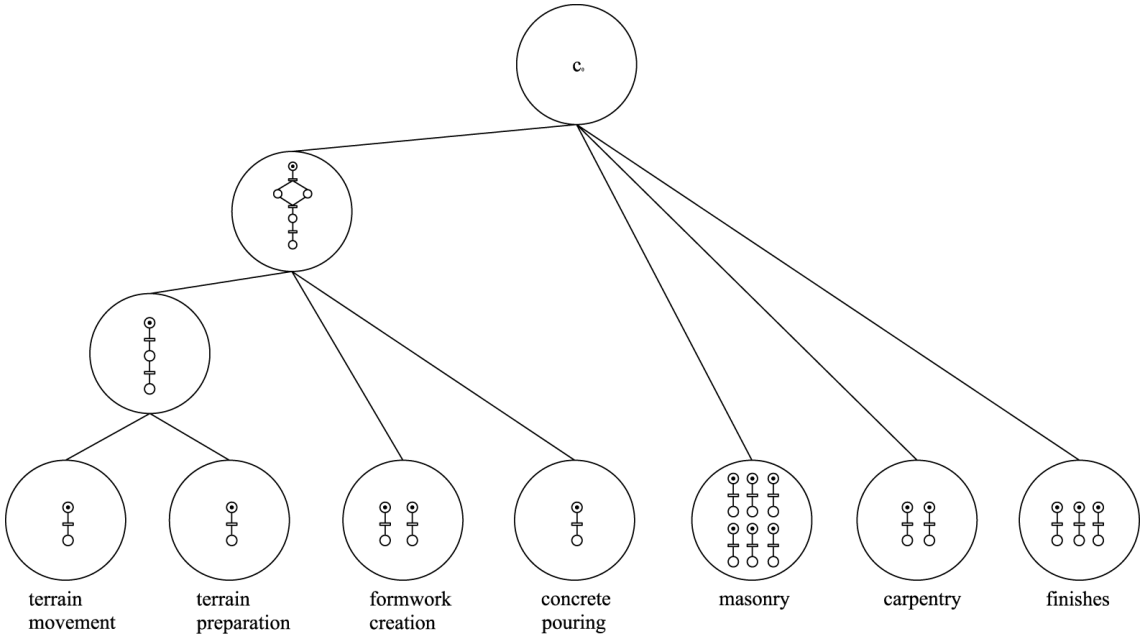


Figure 5.3: Creation of the second complex agent, creation of the structure.

resources is given by $s_{18}^{-} = \sum_{a \in \mathcal{A}_{pet}} \bar{s_a}$, such that $a = (id, ib, PN, s_a)$. Let us remind that now we have $\mathcal{A}_{pet} = \{a_5, a_4, a_3, a_{17}\}$. Figure 5.3 represents graphically this petition.

5.5 Third petition: Create the facade

Next, we want to create the facade. We insert a new petition into the system to construct the facade. Let us consider a petition $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$ where:

- $f^u = 2 * x_1 + 1 * x_2 + .3 * x_3 + 1 * x_4 + 1 * x_6 + 1 * x_{13} + 1 * x_{14}$ is the utility function (all values $0 * x_n$ are not represented).
- $\bar{o} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0)$.
- $\mathcal{A}pet = \{\}$ is the set of agents that are used by the petition (empty in the beginning).

The petition sends its messages, receives agents $\mathcal{A}_{pet} = \{a_{12}, a_{13}, a_6, a_7, a_8, a_{18}\}$ and creates a cell and agent a_{19} and inserts it. Figure 5.4 represents the result of this process.

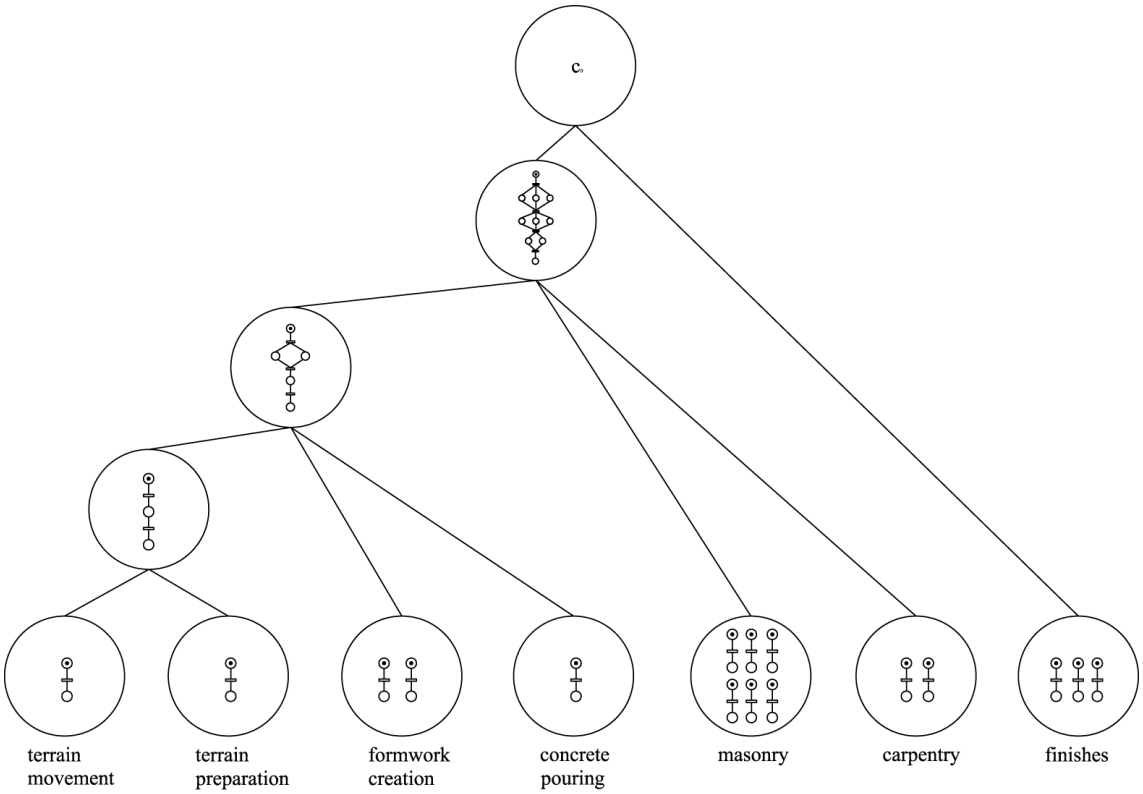


Figure 5.4: Creation of the third complex agent, the facade is finished.

5.6 Fourth petition: Interiors and finishes

Finally we will base ourselves in the agents we were creating above this lines and finish the building by adding the interior walls and the finishes. To do so, let us consider a petition $pet = (f^u, \bar{o}, \mathcal{A}_{pet})$, where:

- $f^u = 2 * x_1 + 1 * x_2 + .3 * x_3 + 1 * x_4 + 1 * x_6 + 1 * x_{15} + 1 * x_{16} + 2 * x_{17} + 1 * x_{18}$ is the utility function (all values $0 * x_n$ are not represented).
- $\bar{o} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0)$.
- $\mathcal{A}pet = \{\}$ is the set of agents that are used by the petition (empty in the beginning).

The petition sends its messages receives agents $\mathcal{A}_{pet} = \{a_{15}, a_{14}, a_{11}, a_{10}, a_9, a_{19}\}$ and creates a cell and agent a_{20} and inserts it. Agent a_{20} already provides the built house. So, if at any other moment in time we need to construct a house again, we will just need to call upon

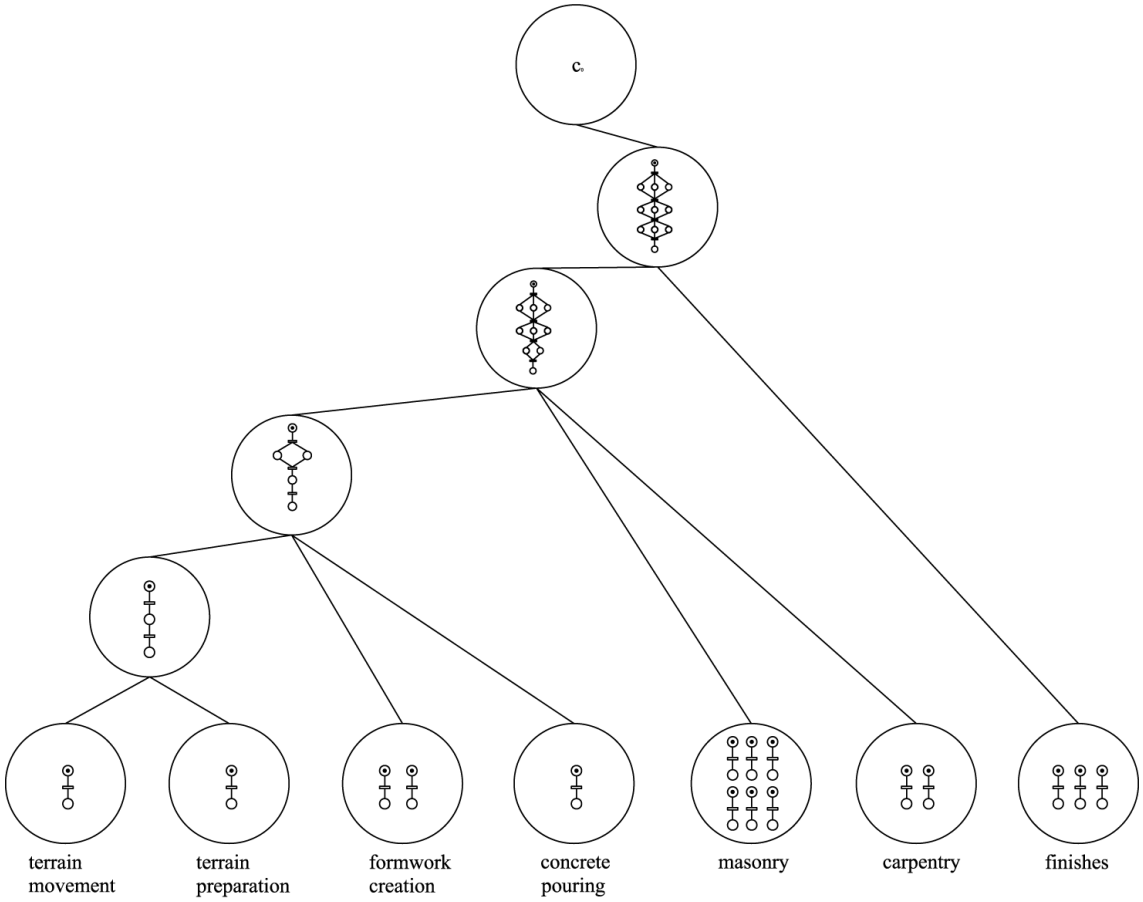


Figure 5.5: Final phase, the house is created.

agent a_{20} and it will make all subsequent calls to every other agent. Figure 5.5 represents the insertion of this last agent.

5.7 Resulting Petri Net from the whole process

If we were to expand subsequently all the complex agents into a complete Petri Net of the whole process we would obtain the Petri Net graphically depicted in Figure 5.6. This expanded version is equivalent to the one that agent a_{20} reflects.

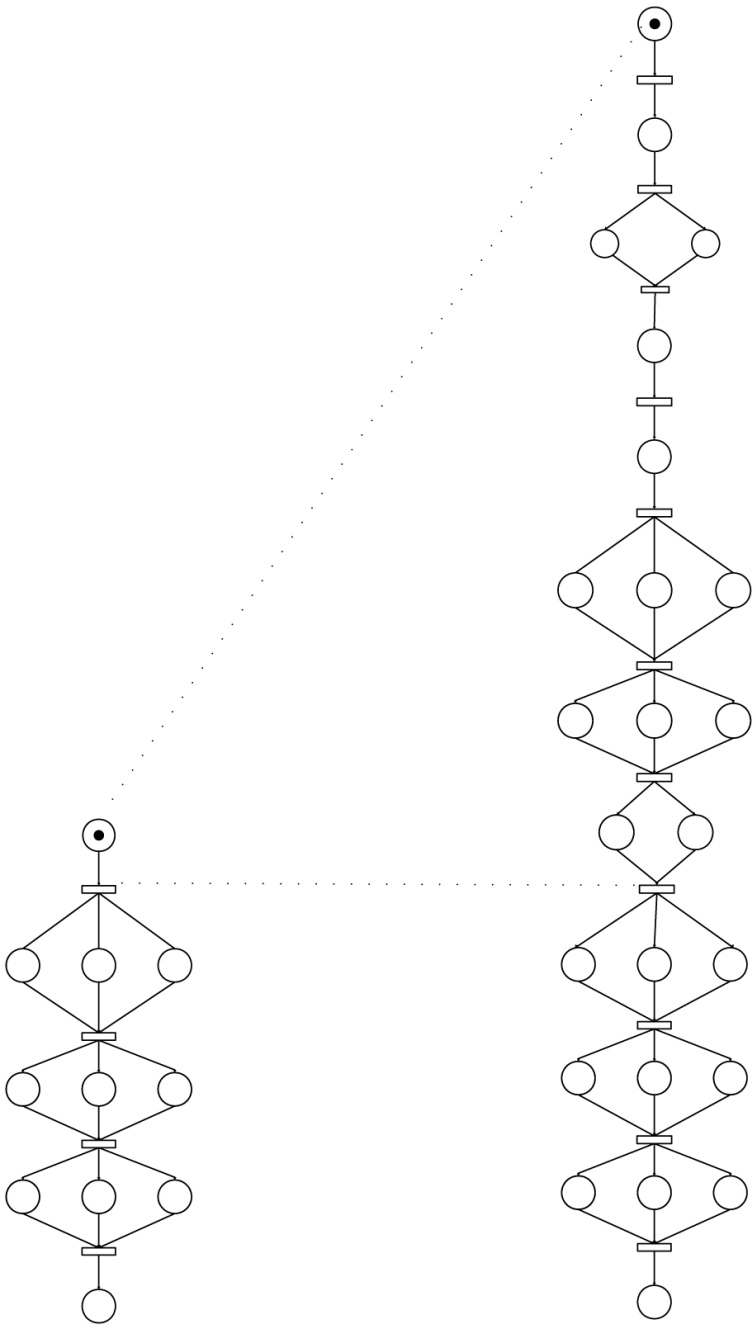


Figure 5.6: Resulting Petri Net

Chapter 6

Conclusions

In this Master Thesis we have presented a formalism to represent complex hierarchical systems where tasks can be distributed and/or *subcontracted* among agents. We are aware that our formalism is difficult to use since there are a lot of mathematical machinery underlying the definition of our systems. Thus, we have decided to build a tool that fully implements our methodology. In this way, a user of our methodology does not need to pay attention to the formal details and can concentrate on defining the appropriate hierarchical structure.

Our approach allows to model systems that will expand with every use. Declaring all the possible atomic tasks that a system can perform as outputs permits the system to complete any petition that the user can foresee. This is done through recombining atomic agents. We continue to add complex agents in every interaction with the tool. Thus, the system is able to perform more complex tasks with each use, that will not need to be re-computed. As well, the distinction between *atomic* and *complex* agents is fundamental since without it, every behavior of the system would need to be pre-implemented before needing it.

We have presented the formalism in two Conferences [AMN08, AMN09] and it will also appear as a chapter in a book. It has been through working on those papers that we realized some details that, if resolved, would create a more complete and flexible approach. This led to the substitution of paths by Petri Nets and finding a way to automatically create the cell tree. The use of Petri Nets, in the enhanced model, has been a great advance in relation with the published papers, since it has added the possibility of parallelizing tasks (agents). This does not only allow us to shorten execution time, but in some cases even creates new emerging behaviors. Also, the automatic creation of the cell tree, due to the computation of the least upper bound of a cell set, is an advantage not to be diminished. It permits to keep

an order, a conceptual structure, of how agents are inserted into the tree. Thus, with this advance, the system is able to save time in its searches. This feature depends on the way the system sends the messages, through its cells, and from them down to each of their sons. Moreover, keeping close in the hierarchy agents that perform similar tasks is an automated feature. Therefore, this represents an improvement with respect to a manual procedure for the insertion of agents, in which agents could have been inserted anywhere.

There are of course limitations in our approach. One of the biggest drawbacks is the simplification of a world as a vector of resources. In this line, a possible future implementation should base the world representation on a BDI (belief, desire, intention) system, with a modal logic and allowing symbolic representation of the world and user needs. However, until date we have considered this issue outside of the focus of our research, so that we could center on the development of the system itself.

Another line of future work is to adhere executable procedures to atomic agents. Therefore, we might try to define a heuristic that allows the system to check the goodness of composing several agents into another one. This entails giving the system some kind of perception of its environment and, therefore, a way to understand the modifications that it produces. Adding a *curiosity* element would allow the system to recompose the agents in an autonomous fashion without the need for petitions, and then, reconfigure itself, to be ready to handle changes in its environment when they happen.

Bibliography

[AMN08] C. Andrés, C. Molinero, and M. Núñez. A formal methodology to specify hierarchical agent-based systems. In *4th Int. Conf. on Signal-Image Technology & Internet-based Systems, SITIS'08*, pages 169–176. IEEE Computer Society Press, 2008.

[AMN09] C. Andrés, C. Molinero, and M. Núñez. A hierarchical methodology to specify and simulate complex computational systems. In *9th Int. Conf. on Computational Science, ICCS'09, LNCS 5544*, pages 347–356. Springer, 2009.

[BCP05] C. Bernon, M. Cossentino, and J. Pavón. An overview of current trends in european AOSE research. *Informatica*, 29:379–390, 2005.

[Bra87] M.E. Bratman. *Intentions, Plans and Practical Reason.* 1987.

[Bro90] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[Bru91] J.C. Brustoloni. Autonomous agents: Characterization and requirements. Technical Report CMU-CS-91-204, School of Computer Science, Carnegie Mellon University, 1991.

[CL90] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[DJJT01] M. Dastani, N. Jacobs, C.M. Jonker, and J. Treur. Modelling user preferences and mediating agents in electronic commerce. In *Agent Mediated Electronic Commerce, The European AgentLink Perspective, LNCS 1991*, pages 163–193. Springer, 2001.

[Dre93] G.L. Drescher. Made-up minds: A constructivist approach to artificial intelligence. *IEEE Expert*, 8(6):76–78, 1993.

[ES89] E.A. Emerson and J. Srinivasan. Branching time temporal logic. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354*, pages 123–172. Springer, 1989.

[FG96] S. Franklin and A.C. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *3rd Workshop on Intelligent Agents: Agent Theories, Architectures, and Language, ATAL'96, LNCS 1193*, pages 21–35. Springer, 1996.

[GHH01] B. Geisler, V. Ha, and P. Haddawy. Modeling user preferences via theory refinement. In *5th Int. Conf. on Intelligent User Interfaces, IUI'01*, pages 87–90. ACM Press, 2001.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *7th European Conf. on Object-Oriented Programming, ECOOP'93, LNCS 707*, pages 406–431. Springer, 1993.

[HH03] V. Ha and P. Haddawy. Similarity of personal preferences: Theoretical foundations and empirical analysis. *Artificial Intelligence*, 146(2):149–173, 2003.

[Jen00] N.R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177(2):277–296, 2000.

[Kis91] G. Kiss. Variable coupling of agents to their environment: Combining situated and symbolic automata. In *3rd European Workshop on Modeling Autonomous Agents and Multi-Agent Worlds, MAAMAW'91*, pages 231–248. Elsevier, 1991.

[LL32] C.I. Lewis and C.H. Langford. *Symbolic Logic*. The Century Corporation, 1932.

[Lom00] I.A. Lomazova. Nested Petri Nets - a formalism for specification and verification of multi-agent distributed systems. *Fundamenta Informaticae*, 43(1-4):195–214, 2000.

[Lom04] I.A. Lomazova. Communities of interacting automata for modelling distributed systems with dynamic structure. *Fundamenta Informaticae*, 60(1-4):225–235, 2004.

[Lom08] I.A. Lomazova. Nested Petri Nets for adaptive process modeling. In *Pillars of Computer Science, Essays Dedicated to Boris Trakhtenbrot on the Occasion of His 85th Birthday, LNCS 4800*, pages 460–474. Springer, 2008.

[Mae89] P. Maes. The dynamics of action selection. In *11th Int. Joint Conf. on Artificial Intelligence, IJCAI'89*, pages 991–997. Morgan Kaufmann, 1989.

[McC79] J. McCarthy. Ascribing mental qualities to machines. Technical Report 326, Stanford AI Lab, Stanford CA, 1979.

[MNR07] M.G. Merayo, M. Núñez, and I. Rodríguez. Formal specification of multi-agent systems by using EUSMs. In *2nd IPM Int. Symposium on Fundamentals of Software Engineering, FSEN'07, LNCS 4767*, pages 318–333. Springer, 2007.

[Mur89] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[MWG95] A. Mas-Colell, M.D. Whinston, and J.R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[NR01] M. Núñez and I. Rodríguez. `PAMR`: A process algebra for the management of resources in concurrent systems. In *21st IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'01*, pages 169–185. Kluwer Academic Publishers, 2001.

[NRR05a] M. Núñez, I. Rodríguez, and F. Rubio. Formal specification of multi-agent e-barter systems. *Science of Computer Programming*, 57(2):187–216, 2005.

[NRR05b] M. Núñez, I. Rodríguez, and F. Rubio. Specification and testing of autonomous agents in e-commerce systems. *Software Testing, Verification and Reliability*, 15(4):211–233, 2005.

[RC04] G. Rizzolatti and L. Craighero. The mirror-neuron system. *Annual Review in Neuroscience*, 27:169–192, 2004.

[RG91] A.S. Rao and M.P. Georgeff. *Modeling Rational Agents within a BDI-Architecture*. Morgan Kaufmann, 1991.

[SBD98] R. Studer, V.R. Benjamins, and D.Fensel. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25((1–2)):161–197, 1998.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[WC01] M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art. In *1st Int. Workshop on Agent-Oriented Software Engineering, AOSE'00, LNCS 1957*, pages 1–28. Springer, 2001.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[WJK00] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[Woo97] M. Wooldridge. Agent-based software engineering. *IEEE Proceedings on Software Engineering*, 144(1):26–37, 1997.

[ZJW01] F. Zambonelli, N.R. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.

How This API Document Is Organized

This API (Application Programming Interface) document has pages corresponding to the items in the navigation bar, described as follows.

Overview

The Overview page is the front page of this API document and provides a list of all packages with a summary for each. This page can also contain an overall description of the set of packages.

Package

Each package has a page that contains a list of its classes and interfaces, with a summary for each. This page can contain four categories:

- Interfaces (italic)
- Classes
- Enums
- Exceptions
- Errors
- Annotation Types

Class/Interface

Each class, interface, nested class and nested interface has its own separate page. Each of these pages has three sections consisting of a class/interface description, summary tables, and detailed member descriptions:

- Class inheritance diagram
- Direct Subclasses
- All Known Subinterfaces
- All Known Implementing Classes
- Class/interface declaration
- Class/interface description

- Nested Class Summary
- Field Summary
- Constructor Summary
- Method Summary

- Field Detail
- Constructor Detail
- Method Detail

Each summary entry contains the first sentence from the detailed description for that item. The summary entries are alphabetical, while the detailed descriptions are in the order they appear in the source code. This preserves the logical groupings established by the programmer.

Annotation Type

Each annotation type has its own separate page with the following sections:

- Annotation Type declaration
- Annotation Type description
- Required Element Summary
- Optional Element Summary
- Element Detail

Enum

Each enum has its own separate page with the following sections:

- Enum declaration
- Enum description
- Enum Constant Summary
- Enum Constant Detail

Use

Each documented package, class and interface has its own Use page. This page describes what packages, classes, methods, constructors and fields use any part of the given class or package. Given a class or interface A, its Use page includes subclasses of A, fields declared as A, methods that return A, and methods and constructors with parameters of type A. You can access this page by first going to the package, class or interface, then clicking on the "Use" link in the navigation bar.

Tree (Class Hierarchy)

There is a Class Hierarchy page for all packages, plus a hierarchy for each package. Each hierarchy page contains a list of classes and a list of interfaces. The classes are organized by inheritance structure starting with `java.lang.Object`. The interfaces do not inherit from `java.lang.Object`.

- When viewing the Overview page, clicking on "Tree" displays the hierarchy for all packages.
- When viewing a particular package, class or interface page, clicking "Tree" displays the hierarchy for only that package.

Deprecated API

The Deprecated API page lists all of the API that have been deprecated. A deprecated API is not recommended for use, generally due to improvements, and a replacement API is usually given. Deprecated APIs may be removed in future implementations.

Index

The Index contains an alphabetic list of all classes, interfaces, constructors, methods, and fields.

Prev/Next

These links take you to the next or previous class, interface, package, or related page.

Frames/No Frames

These links show and hide the HTML frames. All pages are available with or without frames.

Serialized Form

Each serializable or externalizable class has a description of its serialization fields and methods. This information is of interest to re-implementors, not to developers using the API. While there is no link in the navigation bar, you can get to this information by going to any serialized class and clicking "Serialized Form" in the "See also" section of the class description.

Constant Field Values

The Constant Field Values page lists the static final fields and their values.

This help file applies to API documentation generated using the standard doclet.

Hierarchy For All Packages

Package Hierarchies:
auxiliar, clases, clases.PetriNet, dibujo, hierarchicalagents

Class Hierarchy

- java.lang.Object
  - org.jdesktop.application.AbstractBean
    - org.jdesktop.application.Application
      - org.jdesktop.application.SingleFrameApplication
        - hierarchicalagents.**HierarchicalagentsApp**
    - org.jdesktop.application.View
      - org.jdesktop.application.FrameView
        - hierarchicalagents.**HierarchicalagentsView**
  - integracion.xml.**AccesoXml**
  - clases.**Buffer**
  - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.Container
      - javax.swing.JComponent (implements java.io.Serializable)
        - dibujo.**Vidrio**
      - java.awt.Window (implements javax.accessibility.Accessible)
        - java.awt.Dialog
          - javax.swing.JDialog (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - hierarchicalagents.**HierarchicalagentsAboutBox**
        - java.awt.Frame (implements java.awt.MenuContainer)
          - javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - dibujo.**Ventana**

- clases.PetriNet.**Connection**
- dibujo.**ConstantesDibujo**
- clases.**DireccionesAbsolutas**
- clases.**Fachada**
- clases.**GroupOfAgents**
- clases.**Main**
- clases.PetriNet.**Marking**
- clases.**Message**
- clases.**Mundo**
- clases.**Peticion**
- clases.PetriNet.**PetriNet**
- clases.PetriNet.**Place**
- auxiliar.**Recurso**
- auxiliar.**Recursos**
- dibujo.**RGB**
- java.lang.Thread (implements java.lang.Runnable)
  - clases.**Agente**
  - clases.**Celula**
  - auxiliar.**Hilo**
- clases.PetriNet.**Transition**
- clases.**UtilityFunction**

Packages	
auxiliar	
clases	
clases.PetriNet	
concesionario.integracion.xml	
dibujo	
hierarchicalagents	

Serialized Form

Package dibujo

Class dibujo.Ventana extends javax.swing.JFrame implements Serializable

serialVersionUID: 1L

Serialized Fields

ventana

`javax.swing.JFrame` **ventana**

newContentPane

`javax.swing.JComponent` **newContentPane**

vidrio

`Vidrio` **vidrio**

anchoVentana

`int` **anchoVentana**

altoVentana

`int` **altoVentana**

Class dibujo.Vidrio extends javax.swing.JComponent implements Serializable

serialVersionUID: 1L

Serialized Fields

newContentPane

`javax.swing.JComponent` **newContentPane**

x

`int` **x**

y

`int` **y**

ancho

`int` **ancho**

px

`int` **px**

py

`int` **py**

anchoVentana

`int` **anchoVentana**

altoVentana

`int` **altoVentana**

cont

`int` **cont**

anchoRam

`int` **anchoRam**

saltoFilas

`int` **saltoFilas**

ventana

`Ventana` **ventana**

correccionX

`int` **correccionX**

correccionY

`int` **correccionY**

primeraVez

`boolean` **primeraVez**

escala

`double` **escala**

xInicial

`int` **xInicial**

yInicial

```
int yInicial
```

---

color

```
RGB color
```

---

g

```
java.awt.Graphics g
```

Package hierarchicalagents

Class hierarchicalagents.HierarchicalagentsAboutBox extends javax.swing.JDialog implements Serializable

Serialized Fields

closeButton

```
javax.swing.JButton closeButton
```

Constant Field Values

Contents

clases.*

clases.Message		
public static final int	BROADCAST	1
public static final int	FINISHEDJOB	4
public static final int	REPLIES	2
public static final int	STARTJOB	3

clases

Class Agente

```
java.lang.Object
  └java.lang.Thread
      └clases.Agente
```

All Implemented Interfaces:
java.lang.Runnable

```
public class Agente
```

extends java.lang.Thread

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread
java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler

Field Summary

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary

Agente()
Agente(Recursos transformacion)

Method Summary

void	**crearAgenteAtomico**(PetriNet petriNet)
void	**dibujarAgente**(Celula celula)
void	**ejecuta**()
Buffer	**getBuffer**()
int	**getIndiceAgente**()
PetriNet	**getPetriNet**()
Recursos	**getTransformacionGlobal**()
void	**processMessage**(Message mensaje)
void	**run**()

Methods inherited from class java.lang.Thread

activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getAllStackTraces, getContextClassLoader, getDefaultUncaughtExceptionHandler, getId, getName, getPriority, getStackTrace, getState, getThreadGroup, getUncaughtExceptionHandler, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setDefaultUncaughtExceptionHandler, setName, setPriority, setUncaughtExceptionHandler, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Agente

```
public Agente()
```

Agente

```
public Agente(Recursos transformacion)
```

Parameters:
transformacion -

Method Detail

run

```
public void run()
```

Specified by:

run in interface java.lang.Runnable

Overrides:
run in class java.lang.Thread

---

processMessage

```
public void processMessage(Message mensaje)
```

Parameters:
mensaje - The message read from the input buffer.

---

crearAgenteAtomico

```
public void crearAgenteAtomico(PetriNet petriNet)
```

Parameters:
petriNet -

---

dibujarAgente

```
public void dibujarAgente(Celula celula)
```

Parameters:
celula -

---

ejecuta

```
public void ejecuta()
```

---

getBuffer


```
public Buffer getBuffer()
```

---

getIndiceAgente

```
public int getIndiceAgente()
```

---

getPetriNet

```
public PetriNet getPetriNet()
```

---

getTransformacionGlobal

```
public Recursos getTransformacionGlobal()
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS **NEXT CLASS** **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases

Class Celula

```
java.lang.Object
  └java.lang.Thread
      └clases.Celula
```

All Implemented Interfaces:
java.lang.Runnable

```
public class Celula
```

extends java.lang.Thread

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread
java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler

Field Summary

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary

Celula()
Celula(Celula padre)

Method Summary

void	**anadirAgente**(Agente agente)
void	**anadirAgentes**(java.util.ArrayList<Agente> agentes)
void	**anadirHijo**()
void	**anadirHijo**(Celula celula)
void	**anadirHijos**(java.util.ArrayList<Celula> hijos)
void	**dibujaCelula**()
void	**dibujaConexion**()
void	**run**()
void	**sendMessage**(Message message)

Methods inherited from class java.lang.Thread

activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getAllStackTraces, getContextClassLoader, getDefaultUncaughtExceptionHandler, getId, getName, getPriority, getStackTrace, getState, getThreadGroup, getUncaughtExceptionHandler, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setDefaultUncaughtExceptionHandler, setName, setPriority, setUncaughtExceptionHandler, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Celula

public **Celula**()

Celula

public **Celula**(Celula padre)

Method Detail

run

public void **run**()

Specified by:
run in interface java.lang.Runnable
Overrides:
run in class java.lang.Thread

---

sendMessage

public void **sendMessage**(Message message)

---

dibujaCelula

public void **dibujaCelula**()

---

dibujaConexion

public void **dibujaConexion**()

---

anadirHijo

public void **anadirHijo**(Celula celula)

---

anadirHijo

public void **anadirHijo**()

---

anadirHijos

public void **anadirHijos**(java.util.ArrayList<Celula> hijos)

---

anadirAgente

public void **anadirAgente**(Agente agente)

---

anadirAgentes

public void **anadirAgentes**(java.util.ArrayList<Agente> agentes)

Overview **Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases

Class Mundo

```
java.lang.Object
  └clases.Mundo
```

public class **Mundo**

extends java.lang.Object

Method Summary

void	**anadirRecurso**(java.lang.String nombre, double cantidad)
Celula	**dameC0**()
Ventana	**dameVentana**()
void	**DibujaVentana**()
Celula	**getC0**()
int	**getIndiceAgentes**()
static Mundo	**getInstance**()

static Mundo	**getInstancia**()
Recursos	**getRecursos**()
Ventana	**getVentana**()
void	**intercalarCelula**(Celula padre, Celula hijo)
void	**redibujaMundo**(Celula celulaActual)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

dameVentana

public Ventana **dameVentana**()

redibujaMundo

public void **redibujaMundo**(Celula celulaActual)

dameC0

public Celula **dameC0**()

getInstance

public static Mundo **getInstance**()

DibujaVentana

public void **DibujaVentana**()

anadirRecurso

public void **anadirRecurso**(java.lang.String nombre,
double cantidad)

intercalarCelula

public void **intercalarCelula**(Celula padre,
Celula hijo)

getC0

public Celula **getC0**()

getIndiceAgentes

public int **getIndiceAgentes**()

getInstancia

public static Mundo **getInstancia**()

getRecursos

public Recursos **getRecursos**()

getVentana

public Ventana **getVentana**()

clases

Class Fachada

```
java.lang.Object
  └clases.Fachada
```

```
public class Fachada
```

extends java.lang.Object

Constructor Summary

	Fachada()

Method Summary

void	**addResource**(Recurso recurso)
void	**addResources**(Recursos recursos)
void	**createCell**(Celula father)
void	**createPetition**(Recursos pedido, Recursos addList, UtilityFunction f)
void	**drawWorld**()
static Fachada	**getInstance**()
void	**insertAgent**(Recursos transformacion, Celula celula)
void	**load**(java.lang.String name)
void	**save**(java.lang.String name)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Fachada

```
public Fachada()
```

Method Detail

getInstance

```
public static Fachada getInstance()
```

addResources

```
public void addResources(Recursos recursos)
```


addResource

```
public void addResource(Recurso recurso)
```

createCell

```
public void createCell(Celula father)
```

insertAgent

```
public void insertAgent(Recursos transformacion,
                        Celula celula)
```

createPetition

```
public void createPetition(Recursos pedido,
                           Recursos addList,
                           UtilityFunction f)
```

drawWorld

```
public void drawWorld()
```

save

```
public void save(java.lang.String name)
```


load

```
public void load(java.lang.String name)
```

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases
Class DireccionesAbsolutas

```
java.lang.Object
  └clases.DireccionesAbsolutas
```

```
public class DireccionesAbsolutas
```

extends java.lang.Object

Constructor Summary

`DireccionesAbsolutas()`

Method Summary

void	`anadeAgente(Agente agente, int indice)`
Agente	`dameAgente(int indice)`
static DireccionesAbsolutas	`getInstance()`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DireccionesAbsolutas

```
public DireccionesAbsolutas()
```

Method Detail

getInstance

```
public static DireccionesAbsolutas getInstance()
```

anadeAgente

```
public void anadeAgente(Agente agente,
                        int indice)
```

dameAgente

```
public Agente dameAgente(int indice)
```

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases
Class Buffer

```
java.lang.Object
  └clases.Buffer
```

```
public class Buffer
```

extends java.lang.Object

Constructor Summary

`Buffer(int numSlots)`

Method Summary

Message	`fetch()`
void	`insertarMensaje(Message value)`
boolean	`isEmpty()`
void	`print()`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Buffer

```
public Buffer(int numSlots)
```

Method Detail

isEmpty

```
public boolean isEmpty()
```

insertarMensaje

```
public void insertarMensaje(Message value)
```

fetch

```
public Message fetch()
```

print

```
public void print()
```

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Overview **Package** Class **Use** **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE **NEXT PACKAGE** **FRAMES** **NO FRAMES** **All Classes**

Package clases

Class Summary	
Agente	
Buffer	
Celula	
DireccionesAbsolutas	
Fachada	
GroupOfAgents	
Main	
Message	
Mundo	
Peticion	
UtilityFunction	

Overview **Package** Class **Use** **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE **NEXT PACKAGE** **FRAMES** **NO FRAMES** **All Classes**

Overview **Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**
PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases

Class Mundo

```
java.lang.Object
  └clases.Mundo
```

public class **Mundo**

extends java.lang.Object

Method Summary	
void	anadirRecurso(java.lang.String nombre, double cantidad)
Celula	dameC0()
Ventana	dameVentana()
void	DibujaVentana()
Celula	getC0()
int	getIndiceAgentes()
static Mundo	getInstance()

static Mundo	getInstancia()
Recursos	getRecursos()
Ventana	getVentana()
void	intercalarCelula(Celula padre, Celula hijo)
void	redibujaMundo(Celula celulaActual)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

dameVentana

public Ventana **dameVentana**()

redibujaMundo

public void **redibujaMundo**(Celula celulaActual)

dameC0

public Celula **dameC0**()

getInstance

```
public static Mundo getInstance()
```

DibujaVentana

```
public void DibujaVentana()
```

---

anadirRecurso

```
public void anadirRecurso(java.lang.String nombre,
                          double cantidad)
```

---

intercalarCelula

```
public void intercalarCelula(Celula padre,
                             Celula hijo)
```

---

getC0

```
public Celula getC0()
```

---

getIndiceAgentes

```
public int getIndiceAgentes()
```

---

getInstancia


```
public static Mundo getInstancia()
```

---

getRecursos

```
public Recursos getRecursos()
```

---

getVentana

```
public Ventana getVentana()
```

---

clases

Class Main

```
java.lang.Object
  └clases.Main
```

```
public class Main
```

extends java.lang.Object

Constructor Summary
Main()

Method Summary	
static void	main(java.lang.String[] args)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Main

```
public Main()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Parameters:

args -

---

clases.PetriNet

Class Transition

```
java.lang.Object
  └ clases.PetriNet.Transition
```

```
public class Transition
```

extends java.lang.Object

Constructor Summary

Transition()	
Transition(java.lang.String name, Place place)	

Method Summary

void	**augmentQuantity**()
int	**getCantidadActual**()
java.util.ArrayList<Connection>	**getEntryPoints**()
java.util.ArrayList<Connection>	**getExitPoints**()

java.lang.String	**getName**()
void	**setCantidadActual**(int CantidadActual)
void	**setEntryPoints**(java.util.ArrayList<Connection> entryPoints)
void	**setExitPoints**(java.util.ArrayList<Connection> exitPoints)
void	**setName**(java.lang.String name)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Transition

```
public Transition()
```

Transition

```
public Transition(java.lang.String name,
                  Place place)
```

Method Detail

augmentQuantity


```
public void augmentQuantity()
```

getCantidadActual

```
public int getCantidadActual()
```

getEntryPoints

```
public java.util.ArrayList<Connection> getEntryPoints()
```

getExitPoints

```
public java.util.ArrayList<Connection> getExitPoints()
```

getName

```
public java.lang.String getName()
```

setCantidadActual

```
public void setCantidadActual(int CantidadActual)
```

setEntryPoints

```
public void setEntryPoints(java.util.ArrayList<Connection> entryPoints)
```


setExitPoints

```
public void setExitPoints(java.util.ArrayList<Connection> exitPoints)
```

setName

```
public void setName(java.lang.String name)
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases.PetriNet

Class Place

```
java.lang.Object
  └ clases.PetriNet.Place
```

public class **Place**

extends java.lang.Object

Constructor Summary

Place(java.lang.String name)	
Place(java.lang.String name, Transition transition, Agente agenteACargo, Recursos transformacionRecursos)	

Method Summary

void	**addTransition**(Transition transition, Agente agenteACargo, Recursos transformacionRecursos)
int	**getCantidadCompletadas**()
java.util.ArrayList<Connection>	**getConnections**()
java.lang.String	**getName**()
void	**setCantidadCompletadas**(int CantidadCompletadas)
void	**setConnections**(java.util.ArrayList<Connection> connections)
void	**setName**(java.lang.String name)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Place

public **Place**(java.lang.String name)

Place

public **Place**(java.lang.String name,
Transition transition,
Agente agenteACargo,
Recursos transformacionRecursos)

Method Detail

addTransition

public void **addTransition**(Transition transition,
Agente agenteACargo,
Recursos transformacionRecursos)

getCantidadCompletadas

public int **getCantidadCompletadas**()

getConnections

public java.util.ArrayList<Connection> **getConnections**()

getName

public java.lang.String **getName**()

setCantidadCompletadas

public void **setCantidadCompletadas**(int CantidadCompletadas)

setConnections

public void **setConnections**(java.util.ArrayList<Connection> connections)

setName

public void **setName**(java.lang.String name)

clases.PetriNet

Class PetriNet

```
java.lang.Object
  └clases.PetriNet.PetriNet
```

public class **PetriNet**

extends java.lang.Object

Constructor Summary

	PetriNet()

Method Summary

void	**addPlace**(Place place)
void	**addTransition**(java.lang.String namePlace, java.lang.String nameTransition)
void	**addTransition**(Transition transition)
void	**draw**(Ventana ventana, java.awt.Point inicio)

void	**execute**()
Marking	**getM**()
java.util.ArrayList<Place>	**getP**()
Place	**getPlace**(java.lang.String namePlace)
java.util.ArrayList<Transition>	**getT**()
Transition	**getTransition**(java.lang.String nameTransition)
void	**setInitialPlace**(Place place)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PetriNet

public **PetriNet**()

Method Detail

setInitialPlace

public void **setInitialPlace**(Place place)

execute

public void **execute**()

addPlace

public void **addPlace**(Place place)

addTransition

public void **addTransition**(Transition transition)

addTransition

public void **addTransition**(java.lang.String namePlace,
java.lang.String nameTransition)

getTransition

public Transition **getTransition**(java.lang.String nameTransition)

getPlace

public Place **getPlace**(java.lang.String namePlace)

getM

public Marking **getM**()

getP

public java.util.ArrayList<Place> **getP**()

getT

public java.util.ArrayList<Transition> **getT**()

draw

public void **draw**(Ventana ventana,
java.awt.Point inicio)

Uses of Package clases.PetriNet

Packages that use clases.PetriNet

auxiliar	
clases	
clases.PetriNet	

Classes in clases.PetriNet used by auxiliar

PetriNet

Classes in clases.PetriNet used by clases

PetriNet
Place

Classes in clases.PetriNet used by clases.PetriNet

Connection
Marking

Place
Transition

Hierarchy For Package clases.PetriNet

Package Hierarchies:
All Packages

Class Hierarchy

- java.lang.Object
  - clases.PetriNet.Connection
  - clases.PetriNet.Marking
  - clases.PetriNet.PetriNet
  - clases.PetriNet.Place
  - clases.PetriNet.Transition

Package clases.PetriNet

Class Summary

Connection	
Marking	
PetriNet	
Place	
Transition	

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases.PetriNet

Class Marking

```
java.lang.Object
  └clases.PetriNet.Marking
```

```
public class Marking
```

extends java.lang.Object

Constructor Summary

`Marking()`

Method Summary

java.util.ArrayList<Place>	`getActivePlaces()`
java.util.ArrayList<Transition>	`getActiveTransitions()`

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Marking

```
public Marking()
```

Method Detail

getActivePlaces

```
public java.util.ArrayList<Place> getActivePlaces()
```

getActiveTransitions

```
public java.util.ArrayList<Transition> getActiveTransitions()
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS **NEXT CLASS** **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

clases.PetriNet

Class Connection

```
java.lang.Object
  └clases.PetriNet.Connection
```

```
public class Connection
```

extends java.lang.Object

Constructor Summary

`Connection()`

Method Summary

void	`addTransformations(Recursos transformacion)`
void	`ejecuta()`
int	`getAgenteACargo()`
Transition	`getDestinoA()`
Place	`getDestinoB()`

Place	`getOrigenA()`
Transition	`getOrigenB()`
Recursos	`getTransformacionRecursos()`

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Connection

```
public Connection()
```

Method Detail

ejecuta

```
public void ejecuta()
```

addTransformations

```
public void addTransformations(Recursos transformacion)
```

getAgenteACargo

```
public int getAgenteACargo()
```

getDestinoA

public Transition **getDestinoA**()

getDestinoB

public Place **getDestinoB**()

getOrigenA

public Place **getOrigenA**()

getOrigenB

public Transition **getOrigenB**()

getTransformacionRecursos

public Recursos **getTransformacionRecursos**()

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS **NEXT CLASS** **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

dibujo

Class Vidrio

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └dibujo.Vidrio
```

All Implemented Interfaces:
java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable

public class **Vidrio**

extends javax.swing.JComponent

Clase que representa el dibujo de los nodos en la interfaz gr•fica. Extiende de JFrame

See Also:
Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Nested classes/interfaces inherited from class java.awt.Container

java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JComponent
accessibleContext, listenerList, TOOL_TIP_TEXT_KEY, ui, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Vidrio(Ventana ventana)
Vidrio(Ventana ventana, double escala, int xInicial, int yInicial)
Vidrio(Ventana ventana, int ancho1, int alto1)

Method Summary

void	**borra**()

void	**dibujaCirculo**(java.awt.Point punto, int radio)
void	**dibujaCirculo**(java.awt.Point punto, int radio, RGB color)
void	**dibujaCirculoRelleno**(java.awt.Point punto, int radio)
void	**dibujaCirculoRelleno**(java.awt.Point punto, int radio, RGB color)
void	**dibujaLinea**(java.awt.Point puntoIni, java.awt.Point puntoFini)
void	**dibujaLinea**(java.awt.Point puntoIni, java.awt.Point puntoFini, RGB color)
void	**dibujaNodo**(java.awt.Graphics g) Dibuja un
void	**dibujaRectangulo**(java.awt.Point puntoIni, int width, int height)
void	**dibujaRectangulo**(java.awt.Point puntoIni, java.awt.Point puntoFini)
void	**dibujaTexto**(java.awt.Point punto, java.lang.String texto)
void	**dibujaTextoPosicionAbsoluta**(java.lang.String texto, int x, int y)
void	**paintComponent**(java.awt.Graphics g)

Methods inherited from class javax.swing.JComponent

addAncestorListener, addNotify, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getAccessibleContext, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBaseline, getBaselineResizeBehavior, getBorder, getBounds, getClientProperty, getComponentGraphics, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap, getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPopupLocation, getPreferredSize, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getUIClassID, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getVisibleRect, getWidth, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isOpaque, isOptimizedDrawingEnabled, isPaintingForPrint, isPaintingTile, isRequestFocusEnabled, isValidateRoot, paint, paintBorder, paintChildren, paintImmediately, paintImmediately, paramString, print, printAll, printBorder, printChildren, printComponent, processComponentKeyEvent, processKeyBinding, processKeyEvent, processMouseEvent, processMouseMotionEvent, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeNotify, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setComponentPopupMenu, setDebugGraphicsOptions, setDefaultLocale, setDoubleBuffered, setEnabled, setFocusTraversalKeys, setFont, setForeground, setInheritsPopupMenu, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setUI, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update, updateUI

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, getMousePosition, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusCycleRoot, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet,

isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Vidrio

```
public Vidrio(Ventana ventana)
```

Vidrio

```
public Vidrio(Ventana ventana,
              double escala,
              int xInicial,
              int yInicial)
```

Vidrio

```
public Vidrio(Ventana ventana,
              int ancho1,
              int alto1)
```

Method Detail

paintComponent

```
public void paintComponent(java.awt.Graphics g)
```

Overrides:
paintComponent in class javax.swing.JComponent

---

dibujaNodo

```
public void dibujaNodo(java.awt.Graphics g)
```

Dibuja un

Parameters:
g -
color -

---

borra

```
public void borra()
```

---

dibujaLinea

```
public void dibujaLinea(java.awt.Point puntoIni,
                        java.awt.Point puntoFini)
```

dibujaLinea

```
public void dibujaLinea(java.awt.Point puntoIni,
                        java.awt.Point puntoFini,
                        RGB color)
```

dibujaRectangulo

```
public void dibujaRectangulo(java.awt.Point puntoIni,
                             java.awt.Point puntoFini)
```

dibujaRectangulo

```
public void dibujaRectangulo(java.awt.Point puntoIni,
                             int width,
                             int height)
```

dibujaTextoPosicionAbsoluta

```
public void dibujaTextoPosicionAbsoluta(java.lang.String texto,
                                        int x,
                                        int y)
```

dibujaTexto

```
public void dibujaTexto(java.awt.Point punto,
                        java.lang.String texto)
```


dibujaCirculo

```
public void dibujaCirculo(java.awt.Point punto,
                          int radio)
```

dibujaCirculo

```
public void dibujaCirculo(java.awt.Point punto,
                          int radio,
                          RGB color)
```

dibujaCirculoRelleno

```
public void dibujaCirculoRelleno(java.awt.Point punto,
                                 int radio)
```

dibujaCirculoRelleno

```
public void dibujaCirculoRelleno(java.awt.Point punto,
                                 int radio,
                                 RGB color)
```


dibujo

Class Ventana

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
                      └dibujo.Ventana
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class Ventana
```

extends javax.swing.JFrame

Clase que respresenta el dibujo del ventana en la interfaz gr•fica.

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Ventana(java.lang.String title, int ancho, int alto, int posicionX, int posicionY) Constructora que inicializa el dibujo del ventana.

Method Summary

void	**anadirListeners**()
void	**borra**(javax.swing.JComponent componente)
Vidrio	**dameVidrio**()
static Ventana	**getInstancia**(java.lang.String title)
void	**neiniciaVidrio**() Metodo para pintar una nueva habitaci•n.
void	**quitaVentana**() M•todo para borrar la ventana de la interfaz gr•fica.

Methods inherited from class javax.swing.JFrame
addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Frame

addNotify, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setResizable, setState, setTitle, setUndecorated

Methods inherited from class java.awt.Window
addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, setVisible, show, toBack, toFront

Methods inherited from class java.awt.Container
add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener,

setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component
action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener,

removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer
getFont, postEvent

Constructor Detail

Ventana

```
public Ventana(java.lang.String title,
               int ancho,
               int alto,
               int posicionX,
               int posicionY)
```

Constructora que inicializa el dibujo del ventana.

Parameters:

`actX` - coordenada x de la habitacion marcada.
`actY` - coordenada y de la habitacion marcada.
`actZ` - coordenada z de la habitaci•n marcada.

Method Detail

getInstancia

```
public static Ventana getInstancia(java.lang.String title)
```

---

anadirListeners

```
public void anadirListeners()
```

---

dameVidrio

```
public Vidrio dameVidrio()
```

---

neiniciaVidrio

```
public void neiniciaVidrio()
```

Metodo para pintar una nueva habitaci•n.

Parameters:

`i` - coordenada x de la habitacion marcada.
`j` - coordenada j de la habitacion marcada.
`k` - coordenada k de la habitacion marcada.
`p` - puerta marcada dentro de la habitaci•n.

---

borra

```
public void borra(javax.swing.JComponent componente)
```

---

quitaVentana

```
public void quitaVentana()
```


M•todo para borrar la ventana de la interfaz gr•fica.

---

dibujo

Class RGB

```
java.lang.Object
  └dibujo.RGB
```

```
public class RGB
```

extends java.lang.Object

Representa el color RGB

Constructor Summary

`RGB(int red, int green, int blue)` Crea un nuevo color RGB a partir de los colores b•sicos pasados como enteros como par•metro

Method Summary

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

RGB


```
public RGB(int red,
           int green,
           int blue)
```

Crea un nuevo color RGB a partir de los colores b•sicos pasados como enteros como par•metro

Parameters:

`red` - Color rojo
`green` - Color verde
`blue` - Color azul

Uses of Package dibujo

Packages that use dibujo

clases	
clases.PetriNet	
dibujo	

Classes in dibujo used by clases

Ventana
Clase que respresenta el dibujo del ventana en la interfaz gr•fica.

Classes in dibujo used by clases.PetriNet

Ventana
Clase que respresenta el dibujo del ventana en la interfaz gr•fica.

Classes in dibujo used by dibujo

RGB
Representa el color RGB

Ventana
Clase que respresenta el dibujo del ventana en la interfaz gr•fica.

Vidrio
Clase que representa el dibujo de los nodos en la interfaz gr•fica.

Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES All Classes

Hierarchy For Package dibujo

Package Hierarchies:
All Packages

Class Hierarchy

- java.lang.Object
  - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.Container
      - javax.swing.JComponent (implements java.io.Serializable)
        - dibujo.**Vidrio**
      - java.awt.Window (implements javax.accessibility.Accessible)
        - java.awt.Frame (implements java.awt.MenuContainer)
          - javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - dibujo.**Ventana**
  - dibujo.**ConstantesDibujo**
  - dibujo.**RGB**

Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES All Classes

Package dibujo

Class Summary

ConstantesDibujo	
RGB	Representa el color RGB
Ventana	Clase que respresenta el dibujo del ventana en la interfaz gr•fica.
Vidrio	Clase que representa el dibujo de los nodos en la interfaz gr•fica.

Overview Package Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES All Classes

dibujo

Class ConstantesDibujo

```
java.lang.Object
  └dibujo.ConstantesDibujo
```

public class **ConstantesDibujo**

extends java.lang.Object

Constructor Summary

ConstantesDibujo()

Method Summary

static int	**centro**()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ConstantesDibujo

public **ConstantesDibujo**()

Method Detail

centro

public static int **centro**()

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS **NEXT CLASS** **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD